Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Parallele Logiksimulation mit dem Simulator dlbSIM – Implementierungsaspekte und Untersuchungen zur Lastbalancierung

Diplomarbeit

Leipzig, 9. Mai 2001

vorgelegt von
Jens Markwardt

Zusammenfassung:
Die vorliegende Arbeit beschreibt im ersten Teil die Integration des auf MVLSIM basierenden, parallelen Logiksimulators dlbSIM in eine industrielle Simulationsumgebung. Anhand von Experimenten wurde die fehlerfreie Arbeitsweise des parallelen Simulators überprüft und untersucht, inwieweit sich eine Senkung der Simulationszeit gegenüber dem sequentiellen MVLSIM bei der Simulation praxisrelevanter Modelle einstellt. Im Rahmen der parallelen Simulation des *S/390*-Prozessors *Monet*, welcher rund 2.7 Millionen Logikbausteine enthält, konnten in ersten Versuchen bereits Speedup-Werte von bis zu 2.5 erzielt werden.
Im zweiten Teil der Arbeit wurden gezielte Untersuchungen zur dynamischen Lastbalancierung des dlbSIM durchgeführt. Für eine vorgegebene, heterogene Testumgebung wurde eine Teststrategie entwickelt, welche die Untersuchung aller Einflußgrößen der Lastbalancierung erlaubt. Im Verlaufe der Experimente hat sich bestätigt, daß das Lastbalancierungsfeature des dlbSIM die Simulationszeit unter Fremdlasteinfluß signifikant verkürzen kann. Des weiteren ist dlbSIM in der Lage, ungünstige initiale Modellverteilungen auf heterogenen Systemen durch Teilmodellverschiebungen zu kompensieren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

Kapitel 1

Einleitung

„The NEW Amiga 4000/30 features a blistering EC68030 processor running at an incredible 25 MHz, and upgradable at a later date to a faster processor. The A4000/30 has a powerful 4MB of RAM expandable to a total of 18MB. Price including 120MB HD ONLY £1039.99 - Satisfaction guaranteed!“

(Werbeanzeige einer Heimcomputerzeitschrift im Juli 1993.)

Heutzutage dürfte eine solche Anzeige beim Leser wohl bestenfalls noch ein Schmunzeln hervorrufen, denn kaum ein Industriezweig macht so große Entwicklungsfortschritte wie die Computerindustrie. Was vor wenigen Jahren noch als „Traummaschine“ galt, genügt heute nicht einmal mehr zum Briefe schreiben.
Mit der stetig wachsenden Komplexität der zu lösenden Problemstellungen in Wissenschaft, Produktion sowie im Dienstleistungsbereich geht auch eine ständige Nachfrage nach mehr Rechenleistung einher. Will man als Hersteller von Computerschaltkreisen mit dieser Entwicklung Schritt halten und sich auf dem Markt behaupten, so sieht man sich gezwungen, in immer kürzeren Zeiträumen immer leistungsfähigere Schaltkreise herzustellen.
Im Bereich der Großrechner beträgt die Zeitspanne bis zum Erscheinen einer neuen Prozessorgeneration momentan etwa 3 Jahre. In diesem Zeitrahmen wurde beispielsweise auch der neue *Power4*-Prozessor von IBM entwickelt. Der mit 1100 MHz getaktete Chip beinhaltet ca. 170 Millionen Transistoren, welche über eine gute Meile Kupfer miteinander verdrahtet sind. Derartige Zahlen lassen bereits die ungeheure Komplexität moderner Prozessordesigns erahnen.
Von entscheidender Bedeutung ist daher die möglichtst frühzeitige Erkennung von Design- und Entwurfsfehlern. Falls ein kritischer Fehler erst in der Fertigungsphase erkannt wird, kann dieser dem Unternehmen ohne weiteres

Verluste in Millionenhöhe verursachen sowie mehrere Monate Entwicklungsarbeit kosten. Daher ist eine Begleitung sämtlicher Entwurfsphasen durch Verifikationsprozesse unumgänglich.

1.1 Logiksimulation

Der traditionelle Weg[1] der Verifikation von VLSI-Schaltkreisen (*Very Large Scale Integration*) ist die *Simulation*. Sie ist ein Hilfsmittel zur Nachbildung und Untersuchung komplexer Systeme anhand eines *Modells*. Ein Modell stellt hierbei stets eine Vereinfachung des realen Systems dar, bildet jedoch in Bezug auf relevante Aspekte das Verhalten desselben genauestens nach. In Abhängigkeit vom gewünschten Vereinfachungsgrad kann die Modellbildung für die Simulation eines VLSI-Entwurfes auf verschiedenen Abstraktionsstufen erfolgen. Das breite Spektrum reicht hierbei von der niedrigsten Stufe, der Nachbildung von Diffusionsprozessen in kristallinen Strukturen (Prozeßebene) bis hin zur Betrachtung von auf verschiedenen Prozessoren laufenden Prozessen (Systemebene), als höchste Stufe. Mit steigender Stufe nimmt der Vereinfachungsgrad des Modells immer weiter zu, was sich äußerlich in einem geringeren Simulationsaufwand niederschlägt.
Im Designprozeß von Mikroprozessoren nimmt seit der Entstehung leistungsfähiger Hardwarebeschreibungssprachen und Systemen zur Logiksynthese die Simulation auf der Abstraktionsstufe der Logikgatter und Flipflops (*Gate-Ebene*) sowie der Register, Zähler und Multiplexer (*Register-Transfer-Ebene*) eine herausragende Stellung ein. Die Simulation auf diesen beiden Ebenen wird auch als *Logiksimulation* bezeichnet.
Für die Logiksimulation komplexer Mikroprozessor-Entwürfe ist die Trennung der rein funktionellen Verifikation von der Timing-Analyse vorteilhaft. Vor dem Hintergrund dieser Arbeit sei im folgenden eine Beschränkung auf funktionelle Verifikationsmethoden gestattet. Die Realisierung funktioneller Designverifikation auf Gate- und Register-Transfer-Ebene erfolgt mit Hilfe *zyklusbasierter Simulatoren*. Ziel der zyklusbasierten Simulation ist die fortlaufende Ermittlung der Signalwerte an den zyklusbegrenzenden Schaltungskomponenten (Latchen). Aufgrund der relativ niedrigen Abstraktionsstufe der Modelle ist die zyklusbasierte Simulation jedoch sehr zeitaufwendig: Die Simulationszeit kann bis zu sieben Größenordnungen über der tatsächlichen Geschwindigkeit der zu simulierenden Schaltung liegen.

[1] Die zweite Möglichkeit der VLSI-Verifikation ist die formale Verifikation, welche erst seit wenigen Jahren zunehmend an Bedeutung im industriellen Designprozeß gewinnt.

1.2 Beschleuniging zyklusbasierter Simulation

Für die Hardwarebeschreibung des bereits erwähnten *Power4*-Prozessors waren knapp 3 Millionen Zeilen VHDL-Code erforderlich. Um die Fehlerfreiheit des Designs sicherzustellen wurden anschließend insgesamt 120 Milliarden Taktzyklen simuliert. In Anbetracht eines derartigen Verifikationsaufwandes stellt sich sogleich die Frage der Beschleunigung von Simulationsläufen. Zahlreiche Bemühungen auf diesem Gebiet konzentrieren sich auf den Einsatz problemspezifischer Datenstrukturen (binäre Entscheidungsbäume), die Verwendung spezieller Hardwarekomponenten (z.B. Emulatoren) und die Realisierung paralleler Simulationsverfahren auf universellen Mehrprozessorsystemen.
Unter den aufgeführten Möglichkeiten erreichen *Emulatoren* bei weitem die höchsten Geschwindigkeiten. Sie werden meist unter Verwendung von FPGAs (*Field Programmable Gate Arrays*) realisiert und erfreuen sich aufgrund der ständig steigenden Kapazität dieser Bausteine wachsender Beliebtheit. Nachteile im Einsatz von Emulatoren liegen einerseits im relativ aufwendigen Modellbildungsprozeß, vor allem aber in der schlechten Beobachtbarkeit des Emulationslaufes.
Eine erfolgversprechende Lösung dieses Problems besteht im kombinierten Einsatz von Emulation und paralleler Simulation: Falls der Emulator ein Fehlverhalten im Inneren des Schaltungsmodells feststellt, könnte ein paralleler Simulator zum schnellen Auffinden der genauen Fehlerursache eingesetzt werden.

1.3 Hintergrund der Arbeit

Die vorliegende Diplomarbeit entstand im Umfeld eines Projektes mit dem Titel „Modellpartitionierung zur parallelen compilergesteuerten Logiksimulation“. Dieses Projekt ist Teil des Schwerpunktprogrammes „Diskrete Algorithmen und ihre Anwendungen“ der Deutschen Forschungsgemeinschaft (DFG) unter Leitung von Prof. Dr. (Ph. D.) T. Lengauer. Gegenstand des Projektes ist die Entwicklung, Analyse und Implementierung von Partitionierungsalgorithmen für industrierelevante Prozessorstrukturen zur Vorbereitung der parallelen, zyklusbasierten Logiksimulation. Ein unmittelbares Anwendungsfeld wurde durch die Entwicklung der Simulatoren *parallelTEXSIM* und *parallelMVLSIM* [Döhler 1996] geschaffen. Sie entstanden in enger Zusammenarbeit mit dem IBM Forschungs- und Entwicklungslabor Böblingen und stellen die Parallelisierung der IBM-internen zyklusbasierten

Simulatoren *TEXSIM* und *MVLSIM* dar.
Gegenstand einer weiteren Arbeit [Löser 1998] war die Entwicklung des parallelen Simulators *dlbSIM*. Hierbei handelt es sich im Prinzip um parallelMVLSIM, welcher um die Fähigkeit der dynamischen Lastbalancierung erweitert wurde. Motiviert durch Anwendungsszenarien mit Fremdlasteinflüssen gestattet dlbSIM eine dynamische Anpassung des Simulationsprozesses an die Gesamtlastsituation der beteiligten Rechnerknoten. Damit eignet sich dlbSIM auch für den Einsatz auf lose gekoppelten Workstationclustern und benötigt für eine optimale Arbeitsweise im Gegensatz zu parallelMVLSIM keinen exklusiven Zugriff auf einen Parallelrechner.

1.3.1 Motivation und Zielstellung

Als praxisrelevantes Einsatzgebiet für den dlbSIM bieten sich langlaufende Power-On-Simulationen, welche einen kompletten Bootprozeß durchlaufen, an. Zwar erreicht dlbSIM hier bei weitem nicht die Leistung eines Emulators, bietet jedoch als Ausgleich die Möglichkeit größerer Nutzerinteraktion. Dank der dynamischen Lastbalancierung könnte dlbSIM sehr preiswert auf bereits vorhandenen Workstations eingesetzt werden und dort im Hintergrund laufen, so daß die Anschaffung eines teuren Parallelrechners nicht zwingend erforderlich ist.
Im Rahmen der vorliegenden Arbeit bestand die Aufgabe, den dlbSIM mitsamt der zugehörigen Partitionierungsumgebung testweise in das Simulationsumfeld von IBM Austin zu integrieren. Anhand anschließender Tests sollte die fehlerfreie Arbeitsweise sowie die Praxistauglichkeit des parallelen Simulators untersucht werden.
Ein weiteres Ziel bestand in der Durchführung von Untersuchungen zur dynamischen Lastbalancierung des dlbSIM. Für eine vorgegebene, heterogene Testumgebung war eine Teststrategie zu entwickeln, welche eine Untersuchung möglichst aller Einflußgrößen der Lastbalancierung erlaubt. In Abhängigkeit der Ergebnisse ist eine spätere Entwicklung spezieller Partitionierungsalgorithmen für heterogene Workstationcluster denkbar.

1.3.2 Gliederung der weiteren Kapitel

Die Arbeit ist in folgende Kapitel unterteilt:

Kapitel 2 stellt in Kurzform alle Begriffe, Simulatoren und Partitionierungstools vor, die im Rahmen eines parallelen Simulationslaufes mit dlbSIM von Bedeutung sind.

Kapitel 3 beschreibt die Arbeiten zur Integration des dlbSIM in die Simulationsumgebung von IBM Austin. Zum Abschluß werden die Ergebnisse der dortigen experimentellen Untersuchungen präsentiert.

Kapitel 4 ist der näheren Untersuchung der Möglichkeiten des dlbSIM zur dynamischen Lastbalancierung gewidmet. Anfangs werden knapp einige Implementierungsaspekte der dynamischen Lastbalancierung erläutert. Der Hauptteil des Kapitels hat dann die Beschreibung der experimentellen Untersuchungen zur Lastbalancierung zum Inhalt.

Kapitel 5 beinhaltet eine Zusammenfassung und Auswertung der Ergebnisse.

1.4 Danksagung

An erster Stelle gilt mein besonders herzlicher Dank Herrn Dr. K. Hering für die Vergabe dieses anspruchsvollen und interessanten Diplomarbeitsthemas und die freundliche Betreuung seit Beginn meines Projekteintritts. Auch allen anderen am Forschungsprojekt Beteiligten, insbesondere den Herren R. Reilein, J. Löser, S. Trautmann und D. Lucke, möchte ich für ihre fachliche Unterstützung und Hilfe danken.
Des weiteren bedanke ich mich bei den Herren Dr. W. Rösner und J. Long von der IBM in Austin für ihre Unterstützung und Betreuung während meines Werkstudentenaufenthaltes.
Nicht zuletzt möchte ich auch Herrn A. Bluhm für das Programm *loadnet* sowie für die Hilfe in allen UNIX-Fragen danken.

Kapitel 2

Kurzdarstellung der Simulationswerkzeuge

Dieses Kapitel stellt in Kurzform alle Simulatoren und Partitionierungstools vor, die im Rahmen eines parallelen Simulationslaufes mit dlbSIM von Bedeutung sind. In diesem Zusammenhang wird sowohl auf die prinzipielle Arbeitsweise der Programme als auch auf die Bedeutung wichtiger Begriffe eingegangen.

2.1 Die sequentiellen Logiksimulatoren *TEXSIM* und *MVLSIM*

TEXSIM (*Texas Simulator*) ist ein von IBM Austin entwickelter, firmeninterner Logiksimulator. Er wurde von IBM und Motorola erfolgreich bei der Entwicklung führender Mikroprozessorarchitekturen wie *Power2*, *S/390* und *PowerPC* eingesetzt. TEXSIM simuliert zweiwertige synchrone Logikmodelle auf Gate- und Register-Transfer-Ebene. Als Simulationsverfahren kommt der *Clock-Cycle*-Algorithmus zum Einsatz (vgl. nachfolgender Abschnitt).
Als Nachfolger von TEXSIM wurde *MVLSIM* (*Multivalue Simulator*) entwickelt. Wie der Name bereits andeutet, gestattet MVLSIM auch die Auswertung mehrweriger Logik. Ansonsten sind beide Simulatoren in ihrer Arbeitsweise einander sehr ähnlich. MVLSIM ist momentan noch im Einsatz und wurde unter anderem auch bei der Verifikation des *Power4*-Designs eingesetzt. Eine ausführliche Funktionsbeschreibung von MVLSIM ist in [Bergman et al. 1999] zu finden.

2.1.1 Strukturelles Schaltungsmodell

Vereinfacht kann man ein Schaltungsmodell als eine Menge von *Boxen* und *Signalen* auffassen. Eine Box kann hierbei entweder eine zustandslose, logische Funktion (z.B. ein Gatter) realisieren oder aber ein speicherndes Element (z.B. ein Latch) sein. Jede Box kann mehrere Ein- und Ausgabepins besitzen. Die Signale stellen die Verbindungsleitungen zwischen den Boxen dar. Signalanfänge werden als *Sources* und Signalenden als *Sinks* bezeichnet. Ein Signal darf hierbei durchaus mehrere Sources bzw. Sinks besitzen. Der MVLSIM verwendet für die Simulation der Boxen zwei unterschiedliche Zeitmodelle. Kombinatorische Logik wird ohne Verzögerung im sogenannten *Zero-Delay*-Modell evaluiert. Latche hingegen werden als *Unit-Delay*-Boxen betrachtet und einmal pro Taktzyklus durchgeschaltet. Eine umfassende Beschreibung der verschiedenen Zeitmodelle der Logiksimulation ist in [Siedschlag 1998] zu finden. Die kombinatorische Logik eines MVLSIM-Schaltungsmodells muß stets rückkopplungsfrei sein, d.h. Kreise ohne zwischengeschaltete Latche sind nicht gestattet (*synchrones Design*). Auf diese Art läßt sich ein Schaltungsmodell in zwei Teile zerlegen: kombinatorische, zustandslose Logik und Latche. In Abblidung 2.1 ist dieses Prinzip noch einmal verdeutlicht.

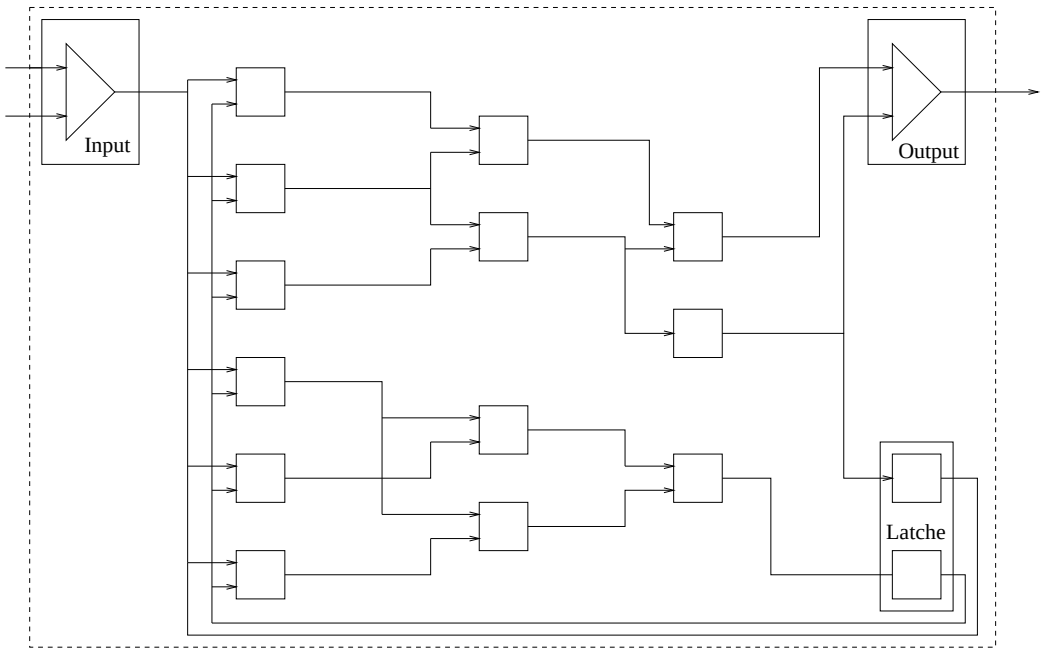


Abbildung 2.1: Strukturelles Schaltungsmodell

Die Simulation eines Taktzyklusses läßt sich aufgrund dieser Zerlegung dann ebenfalls in zwei Phasen durchführen: Im ersten Schritt wird die gesamte

kombinatorische Logik ausgewertet und damit in einen gültigen Zustand überführt. Anschließend werden im zweiten Schritt die Latche durchgeschaltet. Dieser Algorithmus wird als *Clock-Cycle-Algorithmus* bezeichnet.

2.1.2 Erzeugung eines Simulationsmodells

Als Ausgangspunkt für die Simulation dient die Logikbeschreibung in einer Hochsprache, wie zum Beispiel DSL/1 oder VHDL. Ein Pre-Compiler erzeugt daraufhin aus der Hochsprachenbeschreibung ein Zwischenformat, welches als *Proto* bezeichnet wird. Ein Proto enthält die vollständigen Strukturen der Logikbeschreibung und setzt sich aus Signalen, Boxen und Pins zusammen. Im Zusammenhang mit Protos werden die Signale als *Netze* bezeichnet.
Die Verwaltung der Protos erfolgt in einer speziellen Datenbank, der *DA-DB* (*Design Automation Data Base*). Die DA-DB kann mehrere Protos gleichzeitig verwalten und verfügt über ein mächtiges API (*Application Programming Interface*) für die Programmiersprache *C*, über welches ein DA-DB-Client auf die Protos zugreifen kann. Das API erlaubt es, ein Proto in all seinen Eigenschaften zu analysieren und zu verändern. Es ist sogar möglich, aus bereits geladenen Protos neue Protos zu erzeugen und diese anschließend abzuspeichern. Eine Beschreibung des gesamten DA-DB APIs ist in [Zike 1992] zu finden. Zur Veranschaulichung der komplexen Proto-Strukturen sowie zur Fehlersuche stehen zudem spezielle Browser zur Verfügung, welche Protos als Schaltplan graphisch auf dem Monitor darstellen können.
Aufgrund der hohen Komplexität vieler Logikbeschreibungen sind die Protos in der Regel hierarchisch aus Funktionseinheiten aufgebaut. Es gibt ein sogenanntes *Top-Level-Proto*, welches über mehrere Hierarchieebenen selbst wieder aus Protos zusammengesetzt ist. In [Reilein 1998] wird am praktischen Beispiel des *S/390* Prozessors *Monet* der Aufbau eines hierarchischen Protos verdeutlicht. Um nun ein solches Proto in ein Simulationsmodell für den MVLSIM zu überführen, muß es zunächst „eingeebnet“ werden. Hierfür dient der DA-DB-Client *dbflat*.
Aus dem flachen Proto kann anschließend mit Hilfe des Modellbau-Clients *mvlbld* ein Modell für den MVLSIM erzeugt werden. Vor dem eigentlichen Modellbau kann *mvlbld* auf Wunsch umfangreiche Optimierungen[1] unter Verwendung von *dslopt* vornehmen, um die Simulation zu beschleunigen. In der Phase des Modellbaus erzeugt *mvlbld* für die einzelnen Elemente des Protos ausführbare Maschinencodesequenzen, aus denen schließlich das fertige

[1] Die Optimierungen beinhalten unter anderem die Vereinfachung boolscher Ausdrücke (z.B. de Morgan'sche Regel, Erkennung von Identitäten) und Netzwerkoptimierungen (z.B. rekursives Entfernen von sink-losen Netzen, Eliminierung redundanter Boxen, Vereinigung von Latchen gleichen Typs).

Modell zusammengesetzt wird. Während eines Simulationslaufes wird dann im wesentlichen der generierte Maschinencode ausgeführt. Daher gehört der MVLSIM zur Klasse der *Compiled-Code*-Simulatoren. Aus diesem Blickwinkel steckt der „Simulator“ gewissermaßen schon im Modell selbst, denn der eigentliche MVLSIM besitzt in seinen Grundzügen lediglich API-Charakter. Dieses Vorgehen erklärt einerseits die hohe Simulationsgeschwindigkeit von MVLSIM, erschwert aber andererseits auch dessen Portierbarkeit, da für jede Architektur eigener Maschinencode erzeugt werden muß. *Mvlbld* ist in der Lage, Code für *IBM RS/6000*, *HP Precision-Architecture* sowie für *Sun Sparc* zu erzeugen. Eine detailierte Beschreibung des Modellbildungsprozesses ist in [Zike 1996] zu finden.
In Abbildung 2.2 ist der gesamte Modellbildungsprozeß noch einmal zusammenfassend graphisch dargestellt.

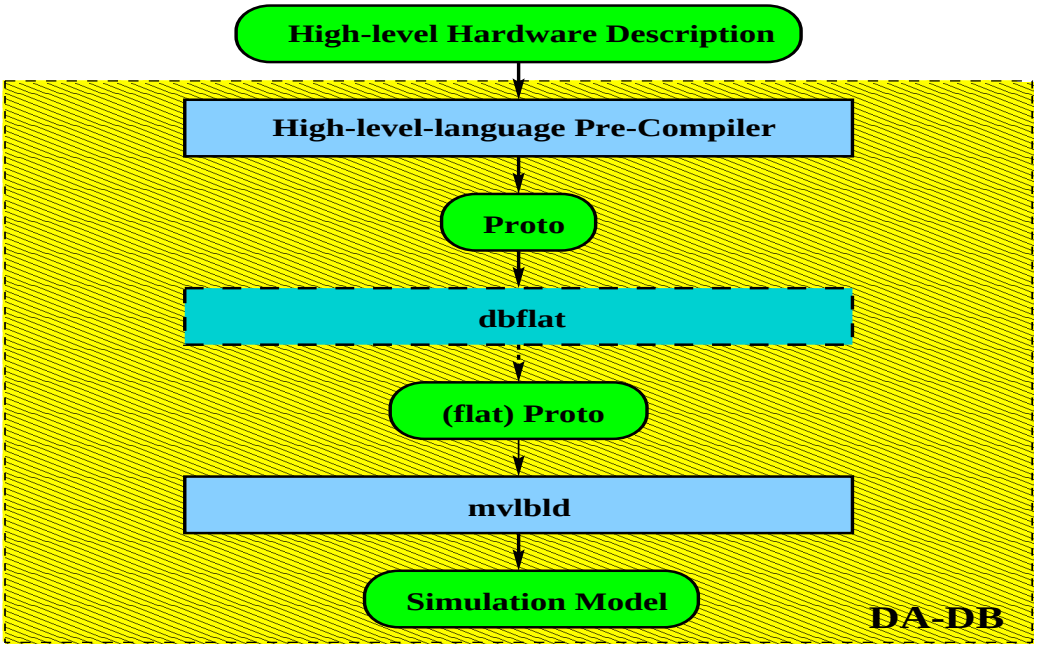


Abbildung 2.2: Schematischer Ablauf der Modellbildung für den MVLSIM

2.1.2.1 Parallel-Instance-Feature

Das *Parallel-Instance-Feature (PIF)* ist eine Methode zur Steigerung der Geschwindigkeit der Logiksimulation. In einem Schaltungsmodell wird es in der Regel mehrere Vorkommen einer bestimmten Teilschaltung oder zumindest eines bestimmten Logikgatters geben. Diese identischen Logikbereiche können sich an beliebigen Stellen der Schaltung befinden und werden daher auch verschiedene Eingangspins besitzen. Ohne Nutzung des PIFs erzeugt

mvlbld für sämtliche Vorkommen identischer Logikbereiche eigenen Code, welcher dann während der Simulation nacheinander ausgeführt wird. Bei Verwendung des PIFs ist eine gleichzeitige Auswertung der identischen Teilschaltungen möglich, wenn die Registerbreite des Prozessors, auf dem die Simulation läuft, größer ist als die Anzahl der Bits, die zur Darstellung von Logikwerten in der entsprechenden Teilschaltung benötigt werden. Im Falle der Simulation mit zweiwertiger Logik lassen sich in einem Byteregister eines Prozessors beispielsweise acht Instanzen eines identischen Logikbereichs berechnen. Die Codererzeugung erfolgt dann ebenfalls nur für eine einzige Instanz.
Ausgangspunkt für die Suche identischer Teilschaltungen in einem Schaltungsmodell ist das hierarchische Proto. Der DA-DB-Client *dbflat* enthält einen Selektionsalgorithmus zur Auswahl der Bereiche, die mit Hilfe des PIFs berechnet werden. Das Ziel des Selektionsalgorithmus besteht in der Auswahl möglichst großer Bereiche für die PIF-Berechnung. In [Reilein 1998] wird etwas näher auf diesen Algorithmus eingegangen. Nachdem der Selektionsalgorithmus die Bereichsauswahl abgeschlossen hat, kann mit dem „Einebnen“ des hierarchischen Protos begonnen werden. In diesem Schritt werden für jeden Logikbereich aus der PIF-Liste spezielle Boxen eingeführt, welche die Zusammenführung (*DSL_JOIN*-Box) und Aufteilung (*DSL_SPLIT*-Box) der Netze für die gleichzeitige Simulation der Instanzen durchführen.

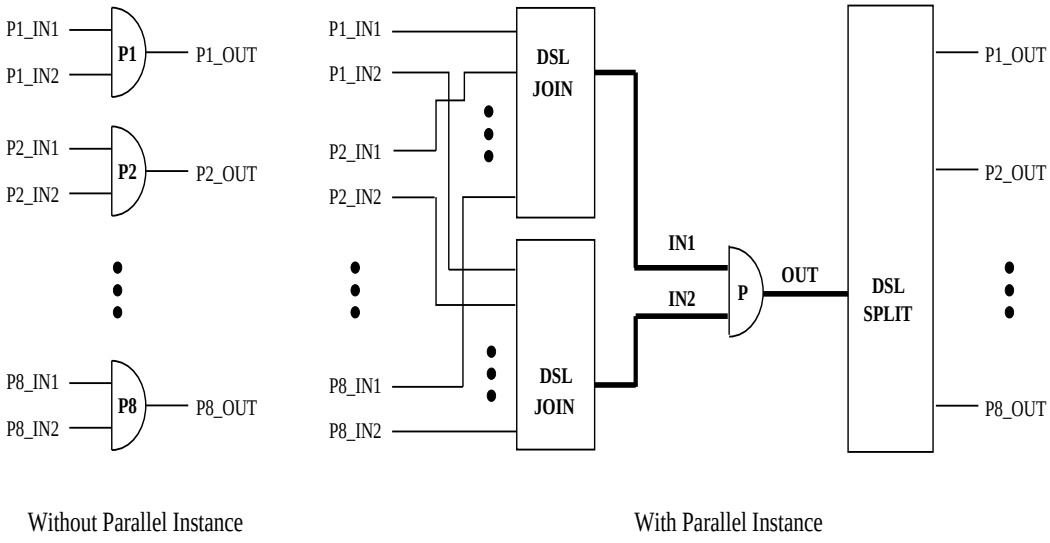


Abbildung 2.3: Beispiel für das Parallel-Instance-Feature

Abbildung 2.3 zeigt ein konkretes Beispiel für die Anwendung des Parallel-Instance-Features: Angenommen, in einem Schaltungsmodell existieren acht identische UND-Gatter mit je zwei Eingängen, wie sie im linken Teil des

Bildes zu sehen sind. Ohne das PIF müßten für deren Evaluierung die in Algorithmus 1 gezeigten Codesquenzen ausgeführt werden:

Algorithmus 1 Logikevaluierung ohne Parallel-Instance-Feature

```
/* UND-Verknüpfung 1 */
LOADBYTE R1, P1_IN1
LOADBYTE R2, P1_IN2
AND R1, R2, R3
STOREBYTE R3, P1_OUT
⋮
/* UND-Verknüpfung 8 */
LOADBYTE R1, P8_IN1
LOADBYTE R2, P8_IN2
AND R1, R2, R3
STOREBYTE R3, P8_OUT
```

Im rechten Teil der Abbildung 2.3 ist die Funktionsweise des PIFs zu erkennen. Hier sind zur Schaltungsauswertung lediglich die folgenden drei Schritte nötig, wie Algorithmus 2 verdeutlicht:

Algorithmus 2 Logikevaluierung mit Parallel-Instance-Feature

```
/* Zusammenführen der Eingänge */
IN1 ⇐ P1_IN1, P2_IN1, ... P8_IN1
IN2 ⇐ P1_IN2, P2_IN2, ... P8_IN2
/* Einmalige UND-Verknüpfung */
LOADBYTE R1, IN1
LOADBYTE R2, IN2
AND R1, R2, R3
STOREBYTE R3, OUT
/* Aufteilung des Ausgänge */
P1_OUT ⇐ OUT(0)
P2_OUT ⇐ OUT(1)
⋮
P8_OUT ⇐ OUT(7)
```

Die Netze *IN1*, *IN2* und *OUT* enthalten als Ergebnis des Zusammenführens jeweils acht einzelne Netze. Zur besseren Unterscheidung von normalen Netzen, werden solche zusammengeführten Netze im folgenden als *PIF-Netze* bezeichnet.

2.1.3 Praktische Durchführung der Simulationsläufe

Der MVLSIM stellt zwei Schnittstellen zur Überwachung und Steuerung der Modellsimulation zur Verfügung: Einen interaktiven Kommandoprozessor (*SUBCMD*-Interface), sowie ein C-API. Die meisten Anwendungen verwenden aus Performancegründen das schnellere sowie flexiblere C-API. Hierzu werden diese zur Laufzeit dynamisch zum MVLSIM hinzugelinkt. Die Kommunikation der Anwendungen mit dem MVLSIM erfolgt nach dem Client-Server-Prinzip, wobei die Anwendungen als Clients fungieren. Die Hauptfunktion der beiden Schnittstellen besteht im Lesen (*Retrieve*) und Schreiben (*Alter*) von Schaltungselementen, sowie in der Simulation von Schaltungszyklen (*Clock*). Die Schaltungselemente werden als *Facilities* bezeichnet. Facilities können Schaltungssignale verkörpernde *Netze*, daraus zusammengesetze *Vektoren* oder *Arrays* sein [Hering et al. 1995]. Die einzelnen Netze eines Vektors werden als *Strands* bezeichnet. Ein Array ist als Vektor von Vektoren aufzufassen.

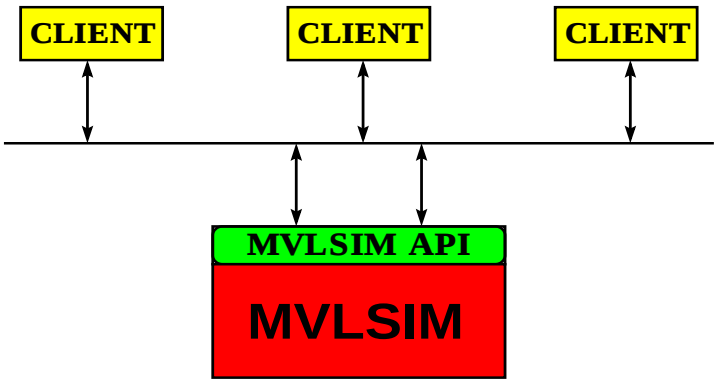


Abbildung 2.4: Client-Server Kommunikation des MVLSIM

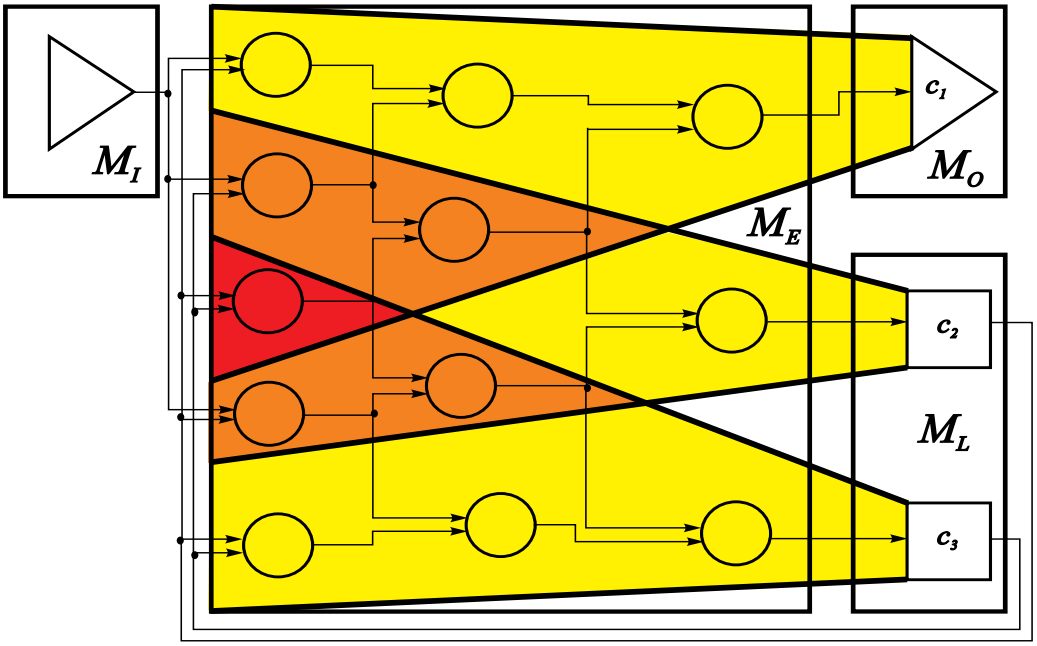
Ein Simulationslauf beginnt stets mit dem Laden und Initialisieren des Modells. Anschließend wird als Client ein Simulationssteuerungsprogramm geladen. Dieses Steuerungsprogramm arbeitet nun eine Reihe sogenannter *Testcases* ab, die zur Überprüfung der Korrektheit des Schaltungsentwurfes dienen. Da der MVLSIM vorwiegend im Prozessordesign eingesetzt wird, bestehen Testcases meist aus Maschinen- oder Microcode-Sequenzen. Nachdem ein Testcase vom Steuerungsprogramm geladen wurde, können die Register und Speicherzellen für diesen Test initialisiert werden (Alter). Anschließend wird die Simulation gestartet und es werden einige Taktzyklen simuliert (Clock), bis eine Test-Ende-Bedingung eintritt. Jetzt kann das Auslesen der Resultate (Retrieve) und deren Überprüfung erfolgen. Da die Designspezifikationen der zu testenden Prozessoren bekannt sind, können Testcases

auch automatisch zufällig mit Hilfe eines Random Test Program Generators (*RTPG*) erzeugt und auf einzelne Rechner eines großen Workstationclusters (von z.B. etwa 500 Maschinen) verteilt werden. Ein spezielles Simulationssteuerungsprogramm namens *RTX* (Random Test Executer) koordiniert dann auf jedem einzelnen Rechner die Simulation der Testcases durch den MVLSIM. Aufgrund des ungünstigen Verhältnisses von realer CPU-Zeit zu simulierter CPU-Zeit (welche bis zu sieben Größenordnungen auseinander liegen können) ist es erforderlich, sämtliche Rechenresourcen optimal auszunutzen. So werden beispielsweise auch die Arbeitsplatzrechner von Mitarbeitern in Leerlaufzeiten (z.B. Mittagspause oder über Nacht) automatisch für die Simulation verwendet.

2.2 Parallelisierungsansatz

2.2.1 Cone-basierte Partitionierung

Grundlage der parallelen Simulation ist eine Partitionierung des strukturellen Schaltungsmodells auf Proto-Ebene. Ein Beispielmodell ist schematisch in Abbildung 2.5 dargestellt. Die grundlegenden Modellbestandteile sind gegeben durch eine Menge von globalen Schaltungseingängen (M_I), globalen Schaltungsausgängen (M_O), kominatorischer Logik (M_E) und Latchen (M_L). Die Partitionierung orientiert sich an sogenannten *Fan-In Cones*. Ein Fan-In-Cone wird aus einem Conekopf, welcher aus M_O oder M_L gewählt wird, sowie aus all denjenigen Logik-Boxen aus M_E gebildet, welche die Möglichkeit besitzen, den Conekopf zu beeinflussen. Die Cones werden dann in geeigneten Gruppen, den sogenannten *Supercones*, zusammengefaßt. Aus den Supercones wiederum entsteht schließlich eine endgültige Zerlegung des Schaltungsmodells in mehrere Blöcke. Diese Zerlegung wird als *Partition* bezeichnet. Die einzelnen Blöcke überführt der Partitionierer wieder in gültige Protos, die anschließend mit dem *mvlbld* in Simulationsmodelle übersetzt und später als Teilmodelle auf den Subsimulatoren simuliert werden können.



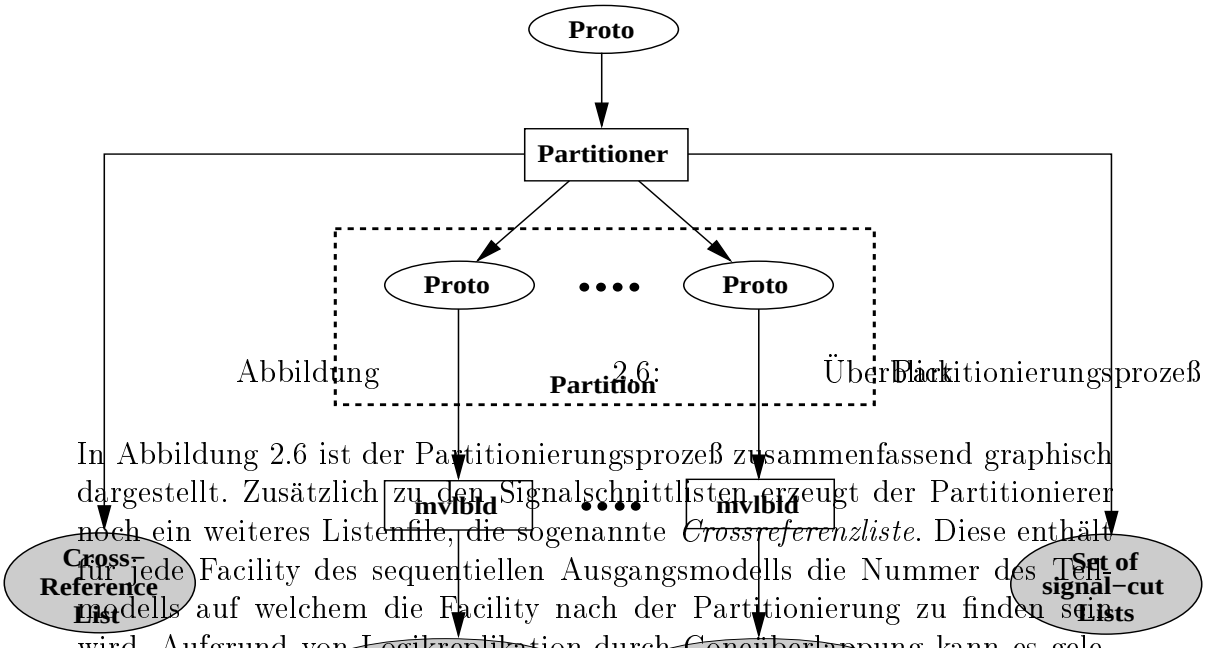


Abbildung 2.6: Überblick Partitionierungsprozeß im

In Abbildung 2.6 ist der Partitionierungsprozeß zusammenfassend graphisch dargestellt. Zusätzlich zu den Signalschnittlisten erzeugt der Partitionierer noch ein weiteres Listenfile, die sogenannte *Crossreferenzliste*. Diese enthält für jede Facility des sequentiellen Ausgangsmodells die Nummer des Teilmodells auf welchem die Facility nach der Partitionierung zu finden sein wird. Aufgrund von Logikreplikation durch Coneüberlappung kann es gelegentlich auch vorkommen, daß sich eine Facility in mehreren Teilmodellen befindet. Die Crossreferenzliste enthält dann für die betroffene Facility mehrere Teilmodellnummern.

2.2.2 Partitionierungsalgorithmen und -umgebungen

Im Rahmen des DFG-Projektes wurden verschiedene Partitionierungsalgorithmen entwickelt. Wie bereits angedeutet wurde, erfolgt die Zerlegung des Schaltungsmodells in zwei Stufen, die als Vorpartitionierung und Partitionierung bezeichnet werden. Die Vorpartitionierung faßt Cones zu Supercones zusammen. Hierfür wurden die Algorithmen *CHAIN*, *STEP*, *LINE* sowie *MUETHU* [Reilein 1998] entwickelt. In der zweiten Stufe werden die Supercones zu Blöcken vereinigt, aus denen sich dann die Simulationsmodelle bilden lassen. Zu diesem Zweck stehen die Algorithmen *SLUCC* [Reilein 1998], *MOCC* (R. Haupt), *nBCC* (K. Hering, A. Voinikonis) [Hering et al. 1995] sowie auch *STEP* zur Verfügung. Es sei erwähnt, daß für eine schnelle Partitionierung auch die direkte Zusammenfassung von Cones zu Blöcken in nur einer Stufe möglich ist. In diesem Fall werden keine Supercones, sondern gleich die fertige Partition erzeugt. Zum Auffinden einer möglichst optimalen Partition bietet es sich an, nach erfolgter Vorpartitionierung gleich mehrere Partitionsläufe in der zweiten Stufe mit unterschiedlichen Parametern und Algorithmen im Wettbewerb durchzuführen und auf diese Weise eine Vielzahl von Partitionen zu erzeugen. Diese Partitionen können dann mit Hilfe der parallelen evolutionären Algorithmen [Schulze 1998] sowie der iterativen Partitionierungsverfahren [Siedschlag 1998] verbessert werden, bis die „beste“ Partition gefunden ist.

2.2.2.1 Die Partitionierungsumgebung *PEnv*

Die *PEnv* wurde von R. Reilein [Reilein 1998] entwickelt. Sie dient der Erzeugung der Datenbasis für die parallelen Simulatoren *parallelTEXSIM*, *parallelMVLSIM* und *dlbSIM*. Die Partitionierungsumgebung arbeitet auf der Grundlage einer einstufigen Modellpartitionierung und besteht aus einzelnen Programmen, welche als Clienten der *DA-DB* bzw. des *TEXSIM/MVLSIM* implementiert sind. Die Ablaufsteuerung erfolgt durch ein UNIX-Shellskript. Dieses Skript erzeugt zunächst eine Batchdatei für die *DA-DB* und startet anschließend die Datenbank zur Abarbeitung derselben. Die *DA-DB* lädt das Schaltungsmodell als Proto in den Speicher, führt ggf. Optimierungen an diesem durch und startet den gewünschten Partitionierungsalgorithmus. Die hierdurch erzeugte Partition wird nun vom Clienten *protobld* unter Verwendung des Schaltungsmodells in mehrere Protos überführt, aus denen wiederum mittels *texbld/mvlbld* die Simulationsmodelle erzeugt werden. Die Erstellung der Crossreferenzliste sowie der Signalschnittlisten übernimmt der Client *filegen*, welcher hierzu ebenfalls die Partition und das originale Schaltungsmodell verwendet. Danach beendet sich die *DA-DB* und

das Shellskript startet zur Nachbearbeitung der Signalschnittlisten den *TEXSIM/MVLSIM* mit dem Simulatorclienten *r2p*. Dies ist notwendig, da das vom parallelen Simulator erwartete Format der Signalschnittlisten Einträge enthält, die im Schaltungsmodell nicht zur Verfügung stehen und die nur vom Simulator selbst ermittelt werden können. Nachdem die Konvertierung der Signalschnittlisten abgeschlossen ist, terminiert das Shellskript und es steht die komplette Datenbasis für einen parallelen Simulationslauf zur Verfügung.

2.2.2.2 Die parallele Partitionierungsumgebung *parallelMAP* – eine Implementierung der *DRIVE*-Umgebung

Die *parallelMAP* (*parallel Model Analysis and Partitioning*) ist eine von H. Hennings [Hennings 1999] entwickelte Test- und Ausführungsumgebung zur Durchführung und Unterstützung der parallelen Modellpartitionierung. Damit stellt *parallelMAP* eine konkrete Implementierung der *DRIVE*-Umgebung (DRIVE = distributed run-time environment) [Hering et al. 1999] dar. *DRIVE* ist in ihrer Anwendung nicht auf die Modellpartitionierung beschränkt, sondern wurde mit dem Ziel entwickelt, Ingenieuren und Wissenschaftlern eine komfortable Umgebung für den Entwurf und die Untersuchung paralleler Algorithmen mit beliebigem Anwendungshintergrund zur Verfügung zu stellen. Als Hardwareresourcen können vernetzte PCs, Workstations und/oder Knoten eines Parallelrechners dienen. *DRIVE* betrachtet diese Resourcen einfach als eine Menge von Prozessoren bzw. darauf ablauffähigen Prozessen, die den Anwendern auf Anforderung dynamisch zugeteilt werden können.
Die modulare Systemarchitektur von *DRIVE* besteht aus den drei Grundkomponenten *Server*, *Client* und *Worker*. Problembezogene *Shared Libraries* und *Datenbanken* vervollständigen die Architektur. Zur Laufzeit existiert in einem *DRIVE*-System genau eine Server-Instanz. Sie bildet die zentrale Komponente und verwaltet die zur Verfügung stehende Menge von Prozessorknoten als potentielle Resourcen zur Ausführung von Rechentasks. Die Anwender können sich durch Starten einer Client-Instanz mit dem Server verbinden. Die Clients stellen eine graphische Benutzerschnittstelle zur Verfügung und unterstützen die Entwicklung paralleler Rechentasks in Form einer einfachen Skriptsprache. Zur Ausführung wird das Skript an den Server übertragen, mit dem der entsprechende Client verbunden ist. Der Server führt zunächst eine Syntaxprüfung des Skripts durch. Anschließend reserviert er die im Skript gewünschte Anzahl von Prozessoren und startet auf jedem von ihnen eine Worker-Instanz. Die Worker bilden dann den Rahmen für die parallele Skriptausführung, indem sie die im Skript festgelegten Programmschritte auf Veranlassung und unter Kontrolle des Servers nacheinander abarbeiten. Die

eigentliche, über Skripte nutzbare Funktionalität der *DRIVE*-Umgebung ist somit in den Worker-Instanzen in Form von internen Funktionen, Datenbanken und Shared Libraries konzentriert. Im Fall der Partitionierungsumgebung *parallelMAP* wird als Datenbank die *DA-DB* verwendet und die dynamisch ladbaren Bibliotheken stellen die verschiedenen Partitionierungsalgorithmen bereit. Zur Veranschaulichung der *DRIVE*-Architektur ist in Abbildung 2.7 ein Server bei der Abarbeitung von zwei Skripten dargestellt.

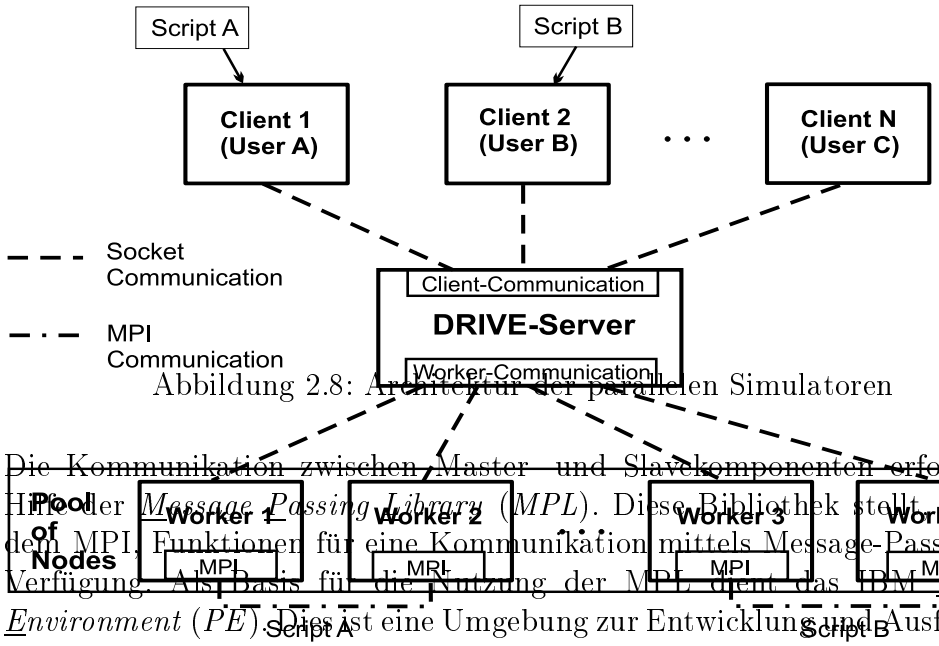
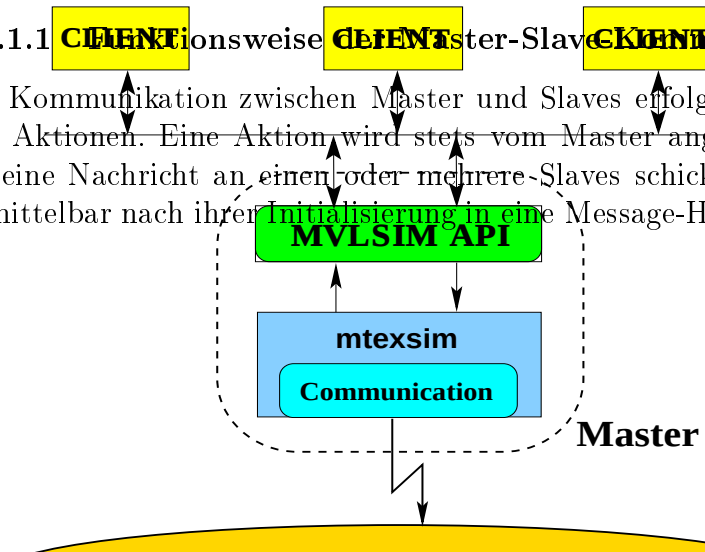


Abbildung 2.8: Architektur der parallelen Simulatoren

Die Kommunikation zwischen Master- und Slavekomponenten erfolgt mit Hilfe der *Message Passing Library* (*MPL*). Diese Bibliothek stellt, ähnlich dem MPI, Funktionen für eine Kommunikation mittels Message-Passing zur Verfügung. Als Basis für die Nutzung der MPL dient das *IBM Parallel Environment* (*PE*). Dies ist eine Umgebung zur Entwicklung und Ausführung paralleler Programme unter dem Betriebssystem *AIX*. Das PE wurde für die *RS/6000*-Architektur entwickelt und ermöglicht die Ausführung paralleler Programme sowohl auf einem losen Verbund von RS/6000-Workstations als auch auf IBM *Scalable POWERparallel (SP)* Systemen. SP-Systeme bestehen aus mehreren *RS/6000*-Knoten, die über einen speziellen *HighPerformanceSwitch* miteinander verbunden sind. Dieser Switch erlaubt sehr hohe Übertragungsraten bei geringen Latenzzeiten. Eine Einführung in die Programmierung unter dem Parallel Environment ist in [PEd 1996] zu finden. Näheres zu den SP-Systemen von IBM kann in [Hennings 1998] nachgelesen werden.

2.3.1.1 Funktionsweise der Master-Slave-Kommunikation

Die Kommunikation zwischen Master und Slaves erfolgt auf der Grundlage von Aktionen. Eine Aktion wird stets vom Master angestoßen, indem dieser eine Nachricht an einen oder mehrere Slaves schickt. Die Slaves treten unmittelbar nach ihrer Initialisierung in eine Message-Handler-Funktion ein,



welche auf eingehende Nachrichten vom Master wartet. Trifft nun eine Nachricht ein, verzweigt der Slave in eine Handler-Routine, um die vom Master spezifizierte Aktion auszuführen. Diese Behandlungsroutine kann nun ihrerseits wieder Kommunikation durchführen, etwa um dem Master auf dessen Anfrage Ergebnisse zurückzusenden (z.B. den Wert einer Facility), oder um Daten mit anderen Slaves auszutauschen (z.B. Austausch der Werte der geschnittenen Netze an den Zyklusgrenzen). Entscheidend ist, daß sämtliche Kommunikation nur im Rahmen der Bearbeitung von Aktionen stattfindet, d.h. ein Slave wird niemals „von sich aus“ mit dem Master oder anderen Slaves in Verbindung treten. Auf diese Weise konnte die Struktur der Kommunikation überschaubar gehalten werden.

2.3.1.2 Parallele Modellsimulation

Ausgangspunkt für die parallele Simulation ist eine korrekte Initialisierung von Master und Slaves. Der Master wird zunächst einige Strukturen zum parallelen Modell anlegen (hierzu gehört auch das Einlesen der Crossreferenzliste) und anschließend die Client-Anwendung laden und dynamisch binden. Ein Slave wird zur Initialisierung das ihm zugeordnete Teilmodell laden sowie die Informationen über die geschnittenen Netze in Form der Signalschnittliste einlesen. Sobald alle Slaves zur Simulation bereit sind übernimmt die Client-Anwendung, wie in einer sequentiellen Simulation, die Kontrolle.
Erhält nun der Master vom Client die Aufforderung, das Modell für eine bestimmte Anzahl n von Takten zu simulieren (*clock n*), so wird er diese an alle Slaves weiterleiten. Die Slaves werden daraufhin n *parallele Clock-Zyklen* durchlaufen. Jeder parallele Clock-Zyklus besteht aus vier Phasen, die im folgenden erläutert werden:

- **Clock:** In diesem Schritt simulieren alle Slaves (gleichzeitig) einen Clock-Zyklus ihres Teilmodells, indem sie den Befehl *„clock 1“* an die ihnen zugrundeliegende MVLSIM-Instanz weitergeben.

- **Get:** Durch den obigen Clock werden die Latche durchgeschaltet und erhalten an ihren Ausgängen neue Werte. Latchausgänge können jedoch mit ausgangsseitig geschnittenen Netzen verbunden sein, welche wieder als Eingänge in andere Teilmodelle führen. *Get* liest die Werte der ausgangsseitig geschnittenen Netze aus dem lokalen Teilmodell aus und schreibt diese in einen Kommunikationsvektor.

- **Transfer:** In diesem Schritt werden die Werte der geschnittenen Netze (d.h. der Kommunikationsvektor) mit den anderen Slaves ausgetauscht. Zum einen werden die mit *Get* ausgelesen Werte an die entsprechenden Rechnerknoten gesendet. Andererseits ist es jedoch auch möglich, daß das eigene lokale Teilmodell eingangsseitig geschnittene Netze besitzt, die ihren Ursprung in Latchausgängen anderer Teilmodelle haben. Auch die Werte diese Netze werden im *Transfer*-Schritt mit dem Kommunikationsvektor von den enstsprechenden Knoten übertragen und können im nachfolgenden *Put*-Schritt ins lokale Teilmodell geschrieben werden. Da alle Slaves am *Transfer*-Schritt teilnehmen, stellt dieser einen Synchronisationspunkt im parallelen Clock-Zyklus dar.

- **Put:** Wie bereits angedeutet, werden hier die empfangenen Werte der eingangsseitig geschnittenen Netze aus dem Kommunikationsvektor ausgelesen und in das eigene Teilmodell geschrieben. Damit ist das Teilmodell zur Abarbeitung eines neuen Clock-Zyklus bereit.

2.3.2 *DlbSIM* – paralleler Logiksimulator mit dynamischer Lastbalancierung

Der von J. Löser [Löser 1998] entwickelte *dlbSIM* stellt den Nachfolger von *parallelMVLSIM* dar. Neben einigen kleineren Modifikationen besteht die bedeutendste Neuerung des dlbSIM in dessen Fähigkeit zur dynamischen Lastbalancierung.
Während der Simulation eines Schaltungsmodells mit parallelMVLSIM gibt es eine bestimmte Anzahl von Subsimulatoren, die alle genau ein Teilmodell des originalen Modells simulieren. Damit existiert eine feste Zuordnung von Teilmodellen zu Subsimulatoren (bzw. Slaves). Für einen „optimalen“ Simulationslauf mit parallelMVLSIM ist daher ein exklusiver Zugriff auf ein SP-System oder ein homogenes Workstation-Cluster erforderlich, da es im Falle von Fremdlasteinwirkung durch andere Nutzer oder bei Verwendung eines heterogenen Netzes von Workstations immer einen „langsamsten“ Rechnerknoten gibt, welcher dann die gesamte Simulation ausbremsen wird.
Der dlbSIM besitzt nun mit seiner integrierten Lastbalancierung die Fähigkeit, Fremdlasteinflüsse sowie Leistungsunterschiede zwischen Rechnern eines heterogenen Clusters durch eine Anpassung des Simulationsprozesses wieder auszugleichen. Während einer dlbSIM-Simulation gibt es wiederum eine bestimmte Zahl von Subsimulatoren, allerdings verfügt jeder Subsimulator jetzt über eine Menge von Teilmodellen. Von diesen potentiell verfügbaren, geladenen Modellen eines Knotens ist zu jedem Zeitpunkt eine gewisse Teilmenge aktiv, d.h. an der Simulation beteiligt. Jedes Teilmodell wird zu

Beginn von mehreren Subsimulatoren geladen, aber es gibt immer nur eine Simulatorinstanz, auf der es gerade aktiv ist (Aktivitätseigenschaft). Unter diesen Voraussetzungen läßt sich die Lastbalancierung einfach über eine Veränderung der Mengen der aktiven Teilmodelle durchführen. Wenn also beispielsweise ein Knoten sehr großen Fremdlasteinflüssen ausgesetzt ist, so kann das Lastungleichgewicht durch Deaktivierung einiger Teilmodelle des belasteten Knotens und anschließende Reaktivierung derselben auf unbelasteten Knoten (welche ebenfalls über die entsprechenden Teilmodelle verfügen müssen) wieder ausgeglichen werden. Das Prinzip der De- und Reaktivierung von Teilmodellen entspricht einer *logischen Modellverschiebung*, da die Teilmodelle nicht wirklich physisch verschoben werden.
Das Lastmanagment des dlbSIM erfolgt parallel zur Simulation auf dem Master. Um eine Entscheidung bzgl. einer Lastverschiebung treffen zu können, muß der Master ein Bild von der aktuellen Lastsituation auf den Slaves haben. Hierzu befragt er defaultmäßig alle 1000 Clock-Zyklen[3] sämtliche Slaves nach deren *simulationsspezifischer Last*. Diese setzt sich aus folgenden zwei Komponenten zusammen:

- der teilmodellbezogenen Evaluationszeit (also diejenige Zeit, welche insgesamt im vergangenen Simulationsintervall allein für *Clock*-Aufrufe benötigt wurde)
- der knotenbezogenen Zeit für das Auslesen und Zurückschreiben der Werte der geschnittenen Netze.

Zusätzlich zur simulationsspezifischen Last senden die Slaves auch ihre, vom Betriebssystem ermittelte, *Load*-Zahl an den Master. Falls ein Knoten derart stark belastet sein sollte, daß er keine aktiven Teilmodelle mehr besitzt, so kann dort auch keine simulationsspezifische Last ermittelt werden. In diesem Fall erhält der Master über den *Load* ein Bild von der vorherrschenden Fremdlastsituation.
Anhand der gesammelten Lastinformationen berechnet der Master nun zunächst für jeden Knoten einen *Lastindex*, der anschließend in einem rekursiven Lastbalancierungsalgorithmus Verwendung findet. Ausgehend von der aktuellen Modellverteilung versucht dieser Algorithmus durch testweises Verschieben von Modellen, eine an die aktuelle Lastsituation angepasste Verteilung der Teilmodelle zu finden. Falls der Algorithmus tatsächlich eine Modellverteilung mit günstigerer Berechnungszeit gefunden hat, müssen die betroffenen Teilmodelle auf ihren bisherigen Knoten deaktiviert und auf

[3] Für die Simulation von 1000 Zyklen wird im Normalfall etwa eine Zeit von einer Minute benötigt. Der Defaultwert läßt sich bei Bedarf über einen Kommandozeilenparameter ändern.

den neuen Knoten aktiviert werden. Bevor die Simulation jedoch weiterlaufen kann, ist es außerdem noch erforderlich, die aktuellen Modellzustände von den alten Knoten auf die neuen zu übertragen. Alles in allem kann eine Modellverschiebung daher eine recht große Simulationsunterbrechung von bis zu zwei Sekunden darstellen. Aus diesem Grund stellt sie eine Ausnahme im normalen Simulationsbetrieb dar und wird nur beim Auffinden einer signifikant besseren[4] Modellverteilung durchgeführt. In Kapitel 4 wird näher auf verschiedene Aspekte der Lastbalancierung eingegangen.

[4]Defaultmäßig muß die neue Berechnungszeit um fünf Prozent besser sein als die bisherige. Auch dieser Wert läßt sich bei Bedarf anpassen.

Kapitel 3

Integration des dlbSIM in eine Simulationsumgebung

Dieses Kapitel beschreibt die im Rahmen eines Werkstudentenaufenthaltes im IBM Entwicklungslabor Austin durchgeführten Arbeiten zur Integration des dlbSIM in die dortige Simulationsumgebung. Im ersten Teil wird auf die Anforderungen seitens der IBM sowie auf aufgetretene Probleme bei der Implementierung und deren Lösung eingegangen. Der zweite Abschnitt geht im Detail auf die durchgeführte Neuimplementierung des Austausches der Werte geschnittener Netze ein. Abschließend werden die Ergebnisse der experimentellen Untersuchungen am „neuen“ dlbSIM vorgestellt.

3.1 Einführungs- und Anpassungsaspekte in Bezug auf die Simulationsumgebung von IBM Austin

3.1.1 Motivation für die Einführung des dlbSIM

Der Großteil der in Austin durchgeführten Simulationsläufe besteht aus verhältnismäßig kleinen Testcases, für welche der sequentielle MVLSIM bestens geeignet ist. Als Einsatzgebiet für den dlbSIM sind hingegen langlaufende Power-On-Simulationen, welche einen kompletten Bootprozeß durchlaufen, denkbar. Gewöhnlich werden derartige Testläufe auf Hardware-Emulatoren wie z.B. *AWAN* durchgeführt. Diese sind zwar um mehrere Größenordnungen schneller als Softwaresimulatoren, müssen aber in der Regel für jede Wertabfrage angehalten werden. Um nun den Geschwindigkeitsgewinn nicht wieder einzubüßen, muß man sich daher bei Emulatoren mit

einem Minimum an abfragbaren Informationen begnügen. Eine Wertabfrage beim dlbSIM ist zwar aufgrund des Netzwerks immer noch langsamer als beim MVLSIM, bleibt aber im Verhältnis viel billiger als bei einem Emulator.
Des weiteren besitzen viele Emulatoren aus Kostengründen Beschränkungen bezüglich der Größe der ladbaren Modelle. Zwar sind moderne Emulatoren durchaus in der Lage, komplette CPU-Kerne inklusive Memory-Controller und I/O-Komponenten zu simulieren, allerdings sind diese auch sehr teuer. Zudem sind Hardware-Emulatoren, ebenfalls aus Kapazitätsgründen, meist nur in der Lage, Modelle mit zweiwertiger Logik auszuwerten. Der dlbSIM besitzt prinzipiell keine Beschränkung in der Modellgröße und unterstützt auch die Auswertung mehrwertiger Logik.
Für Anwendungsfälle, die keine schnellstmögliche Simulation erfordern, steht mit dem dlbSIM somit ein kostengünstiger paralleler Logiksimulator mit dynamischer Lastbalanciering für den Einsatz auf lose gekoppelten Workstations oder SP-Systemen zur Verfügung.

3.1.2 Anforderungen an den dlbSIM seitens der IBM

Um den dlbSIM in der Praxis als Alternative zum MVLSIM einsetzen zu können, müssen folgende Aspekte berücksichtigt werden:

- Der dlbSIM sollte sich möglichst gut in die bestehende Umgebung integrieren, indem er sich in seiner Handhabung idealerweise nicht oder nur kaum von MVLSIM unterscheidet. Besonders wichtig ist hierbei die Bereitstellung eines externen MVLSIM-APIs für die Client-Programme durch den Master.

- Auch der sequentielle MVLSIM ist kein „fertiges“ Produkt sondern wird ständig weiterentwickelt. Um nun diese Änderungen nicht immer „von Hand“ ein zweites Mal an dlbSIM nachvollziehen zu müssen, ist es wünschenswert, die jeweils aktuellen MVLSIM-Quellen auch als Codebasis für die Slaves des dlbSIM zu verwenden. Da der MVLSIM ursprünglich aber nicht vor dem Hintergrund einer späteren Parallelisierung entwickelt wurde, sind Änderungen und Ergänzungen am MVLSIM-Code für dessen Verwendung im dlbSIM unvermeidbar. Diese Änderungen sollten jedoch im Interesse der Wartbarkeit auf ein Minimum reduziert bleiben und besonders sorgfältig kommentiert werden.

- Der dlbSIM sollte intern einen klar strukturierten, modularen Aufbau besitzen. Zugriffe der Slaves auf den integrierten MVLSIM müssen zum größtmöglichen Anteil ebenfalls über das MVLSIM-API erfolgen. Falls die Slaves darüber hinausgehenden Zugriff auf interne Datenstrukturen des MVLSIM benötigen, so sollte dies über ein zusätzliches, klar definiertes API und nicht auf dem schnellen Wege des direkten Speicherzugriffs erfolgen. Auf diese Weise wird eine mögliche Parallelisierung künftiger Logiksimulatoren (z.B. ereignisgesteuerter Natur) auf dlbSIM-Basis wesentlich erleichtert.

3.1.3 Implementierungsprobleme und deren Lösung

3.1.3.1 Problem 1: Slaveseitige Codeverflechtung der parallelen Komponente *stexsim* mit dem zugrundeliegenden MVLSIM

Im Rahmen der Entwicklung des dlbSIM wurde die zu diesem Zeitpunkt (1998) aktuelle MVLSIM-Version als Basis für die Slaves verwendet. Seitdem wurden jedoch in großem Umfang Weiterentwicklungen am MVLSIM vorgenommen, so daß der originale dlbSIM beispielsweise nicht einmal mehr in der Lage war, ein mit dem derzeit aktuellen *mvlbld* erzeugtes Modell einzulesen. Zudem sollte der dlbSIM zum Bau der Slaves ab sofort die jeweils aktuellen MVLSIM-Quellen verwenden, so daß die „Eliminierung“ des alten MVLSIM unvermeidbar war.
Wäre der einzige Kontaktpunkt der Slaves zum darunterliegenden MVLSIM ausschließlich das MVLSIM-API gewesen, dann hätte der MVLSIM-Austausch sicherlich kaum Probleme bereitet. Doch leider erfolgten eine Reihe von Zugriffen auf den MVLSIM unter Umgehung dieses APIs. Außerdem wurden auch Codeänderungen am MVLSIM selbst vorgenommen. Abbildung 3.1 zeigt die enge Verflechtung der parallelen Slave-Komponente *stexsim* mit dem MVLSIM.

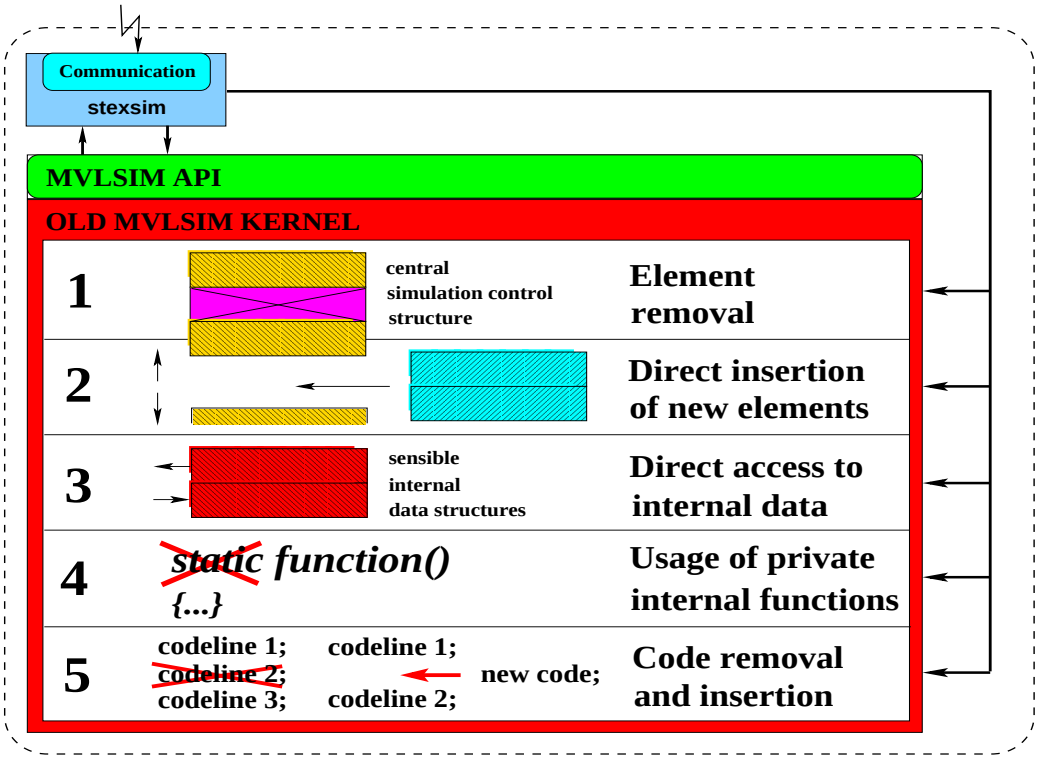


Abbildung 3.1: Bisheriger Zugriff der Slavekomponente *stexsim* auf MVLSIM

1. **Entfernen von Elementen aus der zentralen Simulationskontrollstruktur:** Für die Realisierung der dynamischen Lastbalancierung des dlbSIM lädt eine MVLSIM-Instanz mehrere Teilmodelle. Für jedes Teilmodell existiert dann eine eigene zentrale Simulationskontrollstruktur (*struct SIM_CB*). Einige Elemente dieser Struktur sind aber für alle Teilmodelle gleich (z.B. Pfadnamen, Filepointer usw.), so daß sie aus SIM_CB entfernt und in eine knotenbezogene Struktur (*struct NODE_CB*) unter Kontrolle von *stexsim* verlagert wurden.

2. **Einfügen neuer Elemente in die zentrale Simulationskontrollstruktur:** Es wurden aber auch neue Elemente direkt in SIM_CB eingebaut, da diese von zentraler Bedeutung für die parallele Simulation sind. Dies sind z.B. die jeweilige Teilmodellnummer oder auch Zeiger auf die Signalschnittlisten.

3. **Direkte Lese- und/oder Schreibzugriffe auf MVLSIM-interne Datenstrukturen:** Diese lassen sich grob in zwei Klassen unterteilen: Zum einen erfolgen eine Reihe von Zugriffen auf die zentrale SIM_CB-Struktur (z.B. den Zyklenzähler), was zwar unschön aber noch relativ harmlos ist. Andererseits finden aber aus Gründen der Performance auch Zugriffe auf sensible Datenstrukturen statt (z.B. beim direkten Speicherzugriff auf die geschnittenen Netze über Adresse und Bitposition). Derartige Datenstrukturen können sich leicht von einer Version zur nächsten ändern (was sie auch getan haben) und sollten daher nicht direkt verwendet werden.

4. **Aufruf privater MVLSIM-interner Funktionen:** Einige interne Funktionen des MVLSIM wurden durch Entfernen des *static*-Vorsatzes auch für den *stexsim* nutzbar gemacht (z.B. Funktionen zur Fehlerausgabe, zur Ermittlung vollständiger Pfadnamen oder aber zur Bestimmung der Adresse eines geschnittenen Netzes). Interne Funktionen sind jedoch kein Bestandteil des APIs und können sich in späteren Versionen in ihrem Verhalten ändern oder auch gänzlich verschwinden.

5. **Entfernen und Einfügen von Code in den MVLSIM-Quellen:** Der MVLSIM wurde als rein sequentieller Simulator entwickelt. Daher kam es im Rahmen der Parallelisierung zu Eingriffen in den MVLSIM, die in den meisten Fällen unvermeidbar waren. So wurde beispielsweise neuer Code zur Zeitmessung für den dlbSIM eingefügt. Auch die Signalhandler des MVLSIM wurden entfernt und durch neue ersetzt, damit das *Parallel Environment* überhaupt vom Auftreten eines Signals in Kenntnis gesetzt wird. Es ließen sich noch weitere Beispiele anführen.

Abbildung 3.2 zeigt den neuen Slave im Überblick. Er verwendet nun die aktuellen MVLSIM-Quellen. Außerdem wurde die Verflechtung von *stexsim* und MVLSIM auf ein Minimum reduziert.

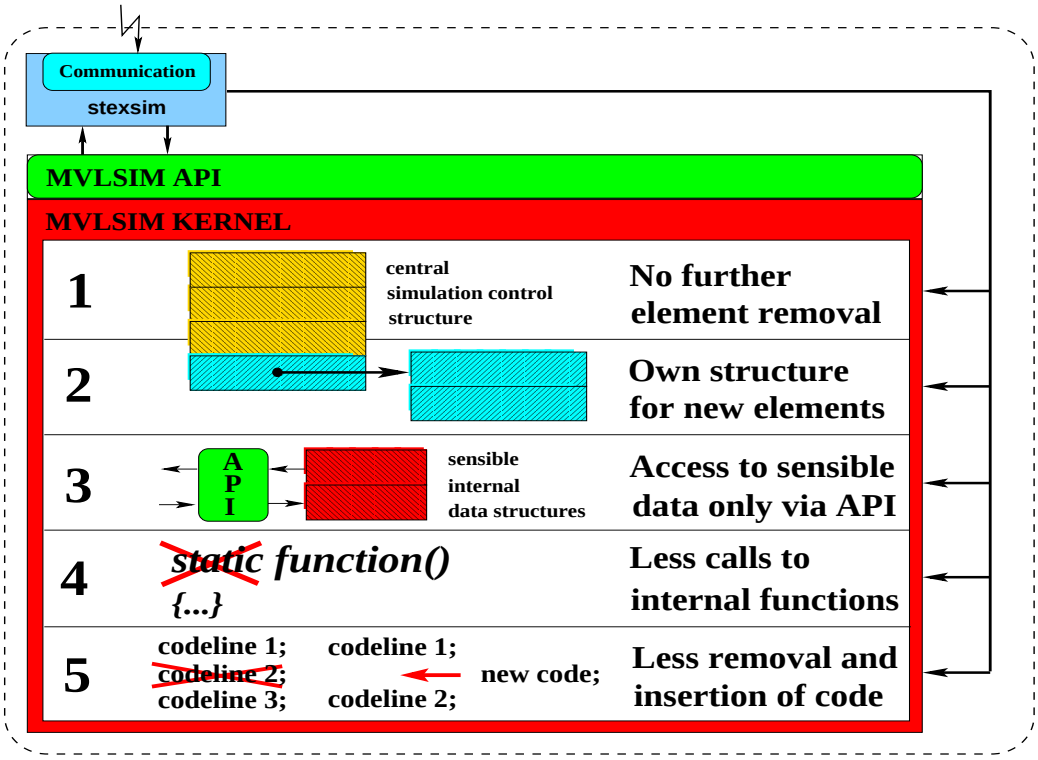


Abbildung 3.2: Neue Zugriffsmethode des *stexsim* auf MVLSIM

1. **Kein weiteres Entfernen von Elementen aus der zentralen Simulationskontrollstruktur:** Da nun für den Slave die aktuellen MVLSIM-Quellen verwendet werden, sind jegliche Änderungen an der SIM_CB-Struktur für die Entwickler des sequentiellen MVLSIM optisch verwirrend[1]. Daher werden für den parallelen Simulator keine Elemente mehr aus SIM_CB entfernt. Die knotenbezogene Struktur NODE_CB bleibt aber weiterhin erhalten. Das redundante Vorkommen der betroffenen Einträge in SIM_CB und NODE_CB wird hierbei in Kauf genommen.

2. **Anlegen einer eigenen Datenstruktur für neue Elemente der zentralen Simulationskontrollstruktur:** Aus Gründen der Übersichtlichkeit werden die neuen Elemente nun in eine eigene Struktur eingefügt, die über einen Pointer am Ende von SIM_CB erreichbar ist.

[1]Sämtliche Änderungen an den MVLSIM-Quellen wurden über *#ifdef SDLBSIM ... #endif* gekapselt und dokumentiert, so daß diese beim Bau des regulären MVLSIM keine Wirkung haben.

3. **Bereitstellung eines neuen APIs (*PS-API = Parallel Simulator-API*) für den Zugriff der Slaves auf sensible MVLSIM-interne Datenstrukturen:** Der Zugriff auf sensible Datenstrukturen erfolgt nun nicht mehr direkt, sondern über ein eigenes PS-API. Dieses API ist als zusätzliche Schnittstelle des MVLSIM für die Slaves des dlbSIM anzusehen. Beispielsweise erfolgt der Zugriff auf die geschnittenen Netze nun nicht mehr direkt, sondern über die Funktionen *ps_getfacdata()* und *ps_putfacdata()*. Die Implementierung neuer Funktionen für das PS-API wird in den meisten Fällen allerdings nur durch die MVLSIM-Entwickler selbst möglich sein, da nur sie die genaue interne Funktionsweise des MVLSIM kennen. Zudem werden sich mit jeder neuen MVLSIM-Version interne Datenstrukturen verändern. Dies kann unter Umständen auch eine Reimplementierung bestimmter PS-API-Funktionen nach sich ziehen. Die Wartung des PS-APIs kann daher ebenfalls nur von IBM selbst vorgenommen werden. Eine Übersicht der bereits implementierten PS-API-Funktionen ist in Anhang B zu finden.

4. **Minimierung der Verwendung privater MVLSIM-Funktionen:** Der Aufruf MVLSIM-interner Funktionen wurde auf ein Minimum reduziert. Einige Aufrufe wurden überflüssig (z.B. Bestimmung der Adresse eines geschnittenen Netzes), andere als Bestandteil von *stexsim* neu implementiert (z.B. Fehlerausgabe). Eine „legale“ Nutzung der wenigen noch verbliebenen privaten MVLSIM-Funktionen wäre durch deren Aufnahme in das PS-API denkbar.

5. **Reduzierung des entfernten und eingefügten Codes in den MVLSIM-Quellen:** In einigen Fällen konnte auf die getätigten Codeänderungen verzichtet werden. Der Großteil der Änderungen ist jedoch für die Funktion des dlbSIM zwingend erforderlich und sollte als fester Bestandteil des MVLSIM beibehalten werden. Es versteht sich von selbst, daß derartige Eingriffe besonders ausführlich im Code kommentiert sind.

3.1.3.2 Problem 2: Aktueller MVLSIM kann nur noch ein Teilmodell verwalten

Die dynamische Lastbalancierung des dlbSIM beruht auf der Fähigkeit des MVLSIM, bis zu acht Modelle gleichzeitig in den Speicher zu laden. Eines dieser Modelle ist stets aktiv und kann simuliert werden. Die Auswahl des aktiven Modells erfolgt über die API-Funktion *txsetid(modelnumber)*. Offenbar hat aber niemand bei IBM von dieser Möglichkeit Gebrauch

gemacht, so daß der Aufwand für die Wartung dieses Features über mehrere MVLSIM-Versionen hinweg nicht mehr gerechtfertigt erschien. Letztendlich kann der MVLSIM in seiner jetzigen Version nur noch ein Teilmodell laden. Der Aufwand für eine Reimplementierung der Verwaltung mehrerer Teilmodelle wird von den MVLSIM-Entwicklern als relativ hoch eingeschätzt und ist daher für die nächste Zeit nicht vorgesehen.
Aus diesem Grund mußte die dynamische Lastbalancierung des dlbSIM vorübergehend deaktiviert werden, so daß sich der dlbSIM nun im Prinzip wie parallelMVLSIM verhält. Sämtliche Datenstrukturen zur dynamischen Lastbalancierung sind aber nach wie vor vorhanden. Die Slaves übertragen sogar noch alle <*dlbinterval*> Zyklen ihre derzeitigen Lastinformationen zum Master. Allerdings werden diese Daten nicht mehr ausgewertet und auch keine Modellverschiebungen initiiert. Der Overhead, welcher durch die Übertragung der Lastinformationen entsteht, kann vernachlässigt werden, da er defaultmäßig nur alle 1000 Zyklen auftritt. Bei Bedarf kann das <*dlbinterval*> natürlich noch weiter erhöht werden.
Sollte der MVLSIM irgendwann wieder die Verwaltung mehrerer Modelle erlauben, so ist für eine Reaktivierung seitens des dlbSIM prinzipiell keine weitere Arbeit nötig[2].

Anmerkung:
Wenn im folgenden der **„neue dlbSIM“** oder **„neue parallele Simulator“** erwähnt wird, so ist damit stets der in Austin überarbeitete dlbSIM mit deaktivierter Lastbalancierung gemeint.

[2]Für eine Modellverschiebung ist die Übertragung des aktuellen Modellzustandes vom „alten“ auf den „neuen“ Knoten erforderlich. Hierzu können die Checkpoint-Funktionen *txsave()* und *txrestore()* verwendet werden. Die MVLSIM-Version des originalen dlbSIM erfaßte mit diesen Funktionen allerdings nicht den kompletten Modellzustand: Die sogenannten *Alters* und *Sticks*, welche das Setzen von Facilities auf bestimmte, feste Werte unabhängig von der treibenden Logik erlauben, waren nicht Bestandteil der Checkpoint-Daten. Der MVLSIM besitzt zwar API-Funktionen zum Setzen von Alters und Sticks, jedoch keine um diese wieder abzufragen. Zu diesem Zweck hat J. Löser die Funktion *determine_sticks()* entwickelt, welche die Alters und Sticks über direkten Zugriff auf interne Datenstrukturen ermitteln konnte. Diese Datenstrukturen haben sich im aktuellen MVLSIM natürlich verändert, so daß *determine_sticks()* nicht mehr funktionieren wird. Glücklicherweise stellt dies kein Hindernis für eine mögliche Reaktivierung der dynamischen Lastbalancierung dar, da bereits die Checkpoint-Funktionen der nächsten MVLSIM-Version Alters und Sticks berücksichtigen werden.

3.1.3.3 Problem 3: Wahl der geeigneten Partitionierungsumgebung

Im Zuge der Auswahl der zu verwendenden Partitionierungsumgebung fiel die Entscheidung zunächst zugunsten der *parallelMAP* aus, da sie eine parallele Modellpartitionierung ermöglicht und eine komfortable graphische Benutzeroberfläche zur Verfügung stellt.
Es hat sich allerdings herausgestellt, daß die Durchführung eines Simulationslaufes bei IBM in der Regel auf Skriptbasis ohne Nutzerinteraktion erfolgt: Das Skript startet die *DA-DB* und veranlasst das Laden des Ausgangsprotos sowie die Erzeugung des Simulationsmodells mittels *mvlbld*. Sobald das Modell existiert, erfolgt die Aktivierung des MVLSIM zur Abarbeitung der Testcases. Zum Abschluß wird der Nutzer über Erfolg oder Mißerfolg der Testcases in Kenntnis gesetzt. Aufgrund dieser Vorgehensweise war die *parallelMAP* mit ihrer interaktiven, graphischen Benutzeroberfläche hier nicht das geeignete Werkzeug zur Modellpartitionierung, so daß die Wahl letztendlich auf die *PEnv* mit ihrer einfachen Skriptumgebung fiel.
Allerdings war die *PEnv* erst nach Durchführung kleinerer Bugfixes und folgender Anpassungen voll einsetzbar:

- **Verwendung der aktuellen PIF-Implementierung beim Bau der Teilprotos:** Die aktuelle Version des Parallel-Instance-Features erlaubt die gleichzeitige Auswertung von bis zu 32 identischen Logikbereichen (bisher waren es maximal acht). Die neuen PIF-Funktionen werden durch die *MVLLIB* zur Verfügung gestellt und sind in [Hidvegi und Shadowen 2000] dokumentiert.

- **Überarbeitung der Erzeugung der Listenfiles:** Die Verwendung der neuen PS-API-Funktionen *ps_getfacdata()* und *ps_putfacdata()* zum Zugriff auf die geschnittenen Netze innerhalb des dlbSIM hat auch eine Änderung des Formats der Signalschnittlisten nach sich gezogen. Der folgende Abschnitt ist speziell diesem Thema gewidmet.

3.2 Neuimplementierung des Signalaustausches über geschnittene Netze

3.2.1 Geschnittene Netze im Detail

Abbildung 3.3 zeigt die vereinfachte Darstellung eines Simulationsmodells, welches sich aus drei Teilmodellen zusammensetzt. Aus Gründen der Übersichtlichkeit wurde jedem Teilmodell nur ein Cone zugrundegelegt. Es ist des weiteren ein geschnittener Vektor V mit vier Strands dargestellt. Er hat seinen Ursprung am Latchausgang (L) von Teilmodell 2 und teilt sich danach auf: Strand 3 führt in Teilmodell 3, der komplette Vektor geht aber gleichzeitig auch in Teilmodell 1.

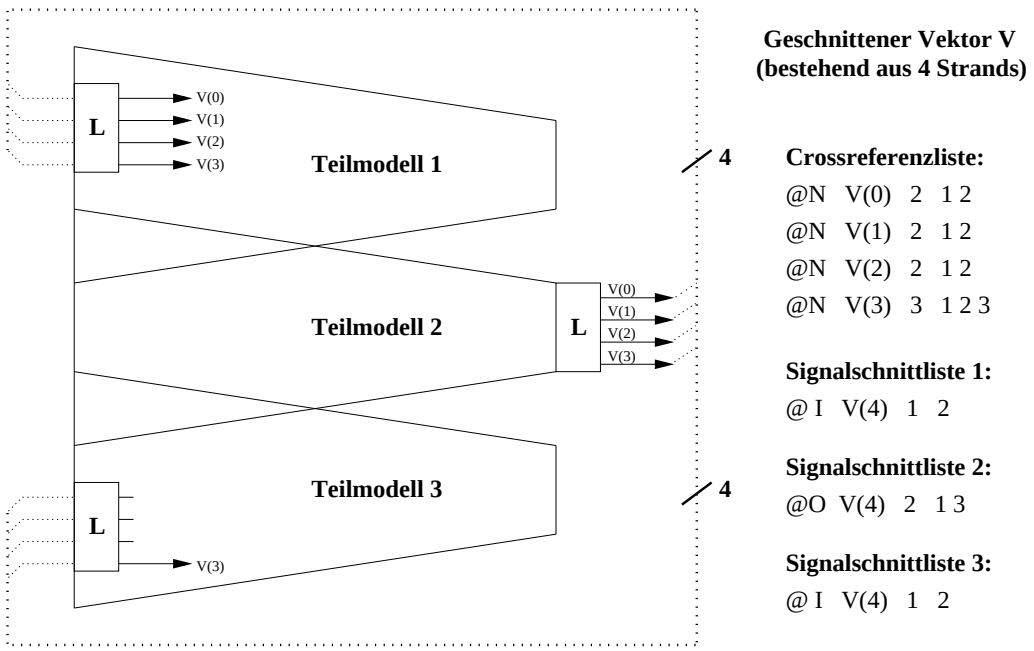


Abbildung 3.3: Beispiel für einen geschnittenen Vektor

Aus Sicht des Simulators gibt es keine geschnittenen Netze oder Vektoren im eigentlichen Sinne. Für ihn sind es lediglich Latche, deren Namen sich aus den Netznamen an den Latchausgängen ergeben. Ein Zugriff auf ein geschnittenes Netz oder einen geschnittenen Vektor entspricht daher intern immer einem Latchzugriff.
Die beiden Latche am Eingang der Teilmodelle 1 und 3 wurden während der Partitionierung „künstlich“ eingefügt, um die eingangsseitig geschnittenen Netze source-seitig abzuschließen, damit diese nicht von *dlsopt* entfernt

werden können. Im Falle eines geschnittenen Vektors werden diese Eingangslatche stets mit voller Bitbreite eingefügt, auch wenn tatsächlich nur 1 Strand benötigt wird (wie bei Teilmodell 3).
Die Übertragung der Werte geschnittener Netze geschieht dann prinzipiell auf folgende Art und Weise:

1. Auslesen der Ausgangslatche am Conekopf.
2. Transport der Netz-Werte über das Netzwerk.
3. Zurückschreiben der Netz-Werte in die Eingangslatche.

3.2.2 Bisherige Methode des Signalaustausches

Wie bereits erwähnt wurde, erfolgte der Zugriff auf die geschnittenen Netze bislang über einen direkten Speicherzugriff auf dasjenige Bit, welches den Wert des entsprechenden Netzes repräsentiert. Für den Austausch der Werte der geschnittenen Netze hatte dieses Vorgehen folgende Konsequenzen:

- Es ist keine Übertragung mehrwehrtiger Netze möglich, da bei dieser Art des Zugriffs nur ein Bit pro Netz zur Verfügung steht. Dies bedeutet, daß für die Simulation nur der zwei-wertige TEXSIM-Mode genutzt werden kann.
- Im Falle eines geschnittenen Vektors muß jeder Strand einzeln ausgelesen und zurückgeschrieben werden.
- Im Falle eines PIF-Netzes (Parallel-Instance-Feature) muß jede Instanz einzeln übertragen werden.

3.2.3 Neue Methode des Signalaustausches

Mit der Implementierung des PS-APIs erfolgt der Zugriff auf die geschnittenen Netze und Vektoren über die Funktionen *ps_getfacdata()* und *ps_putfacdata()*. Diese Funktionen erlauben es, auf Netze und Vektoren in gleicher Weise, unter Angabe des Facility-Index[3], zuzugreifen. Aus diesem Grund werden die geschnittenen Netze und Vektoren im folgenden als *geschnittene Facilities* bezeichnet. Der Zugriff auf die geschnittenen Facilities über die Funktionen *ps_getfacdata()* und *ps_putfacdata()* bringt folgende Vorteile mit sich:

[3]Jeder Facility ist intern ein eindeutiger Index zu deren Identifizierung zugeordnet. Dieser Index wird naheliegenderweise als Facility- oder Simulator-Index bezeichnet. Ein Zugriff auf eine Facility über deren Namen würde zuviel Zeit in Anspruch nehmen.

- Im Multivalue-Mode des dlbSIM werden nun auch mehrwertige Signale übertragen. Es werden dann alle zur Darstellung des gewünschten Logikpegels benötigten Bits ausgelesen und zurückgeschrieben.
- Beim Austausch der Werte geschnittener Vektoren werden stets sämtliche Strands mit einem Zugriff übertragen. Im Beispiel der Abbildung 3.3 würden dann 4 Bits in das Eingangslatch des Teilmodells 3 geschrieben werden, auch wenn das Schreiben von Strand 3 im Prinzip ausreichen würde. Dies ist aber nicht unbedingt als Nachteil anzusehen, da das Schreiben kompletter Datenworte nicht zwangsläufig mehr Zeit in Anspruch nehmen muß als das Schreiben einzelner Bits.
- Falls eine geschnittene Facility das Parallel-Instance-Feature verwendet, so lassen sich alle Instanzen mit einem Mal auslesen und zurückschreiben.

Allerdings darf auch der Nachteil der Funktionen *ps_getfacdata()* und *ps_putfacdata()* an dieser Stelle nicht verschwiegen werden: Sie sind etwas langsamer als die bisherige Methode des direkten Speicherzugriffs, wie die am Ende dieses Kapitels beschriebenen Experimente zeigen werden.

3.2.4 Erläuterungen zu den verschiedenen Listenfiletypen

An dieser Stelle sollen nun die zahlreichen Arten von Listenfiles etwas näher beschrieben werden, da dieses Thema in den bisherigen projektbezogenen Diplomarbeiten zur parallelen Logiksimulation lediglich am Rande angeschnitten wurde. Allerdings sind die in den Listenfiles enthaltenen Datenstrukturen von grundlegender Bedeutung für das Verständnis des Austausches der Werte geschnittener Facilities. Eine vollständige Formatbeschreibung aller Listenfiles mit Beispielen ist in Anhang A zu finden.
Wie bereits erwähnt wurde, existieren zwei grundsätzliche Arten von Listenfiles: Die Crossreferenzliste für den Master und die Signalschnittlisten für die Slaves. Im wesentlichen enthält die Crossreferenzliste Informationen über die Zuordnung der Facilities zu den einzelnen Teilmodellen. Die Signalschnittlisten enthalten nähere Informationen bezüglich der ein- und ausgangsseitig geschnittenen Facilities eines Teilmodells. Von beiden Listenfiletypen existieren Klartext- und Binärversionen. Die binären Listen enthalten die Informationen bereits in der Form, wie der Simulator sie erwartet. Damit sind keine umständlichen Datenkonvertierungen nötig und die mitunter sehr großen Listenfiles können schnellstmöglich eingelesen werden. Für den täglichen Einsatz des Simulators werden daher die binären Listenfiles Verwendung finden.

Die Klartextversionen hingegen können bei der Wartung der Partitionierungsumgebung und des parallelen Simulators eine große Hilfe sein. Zunächst sollen zur Erläuterung der Klartextfiles die Listeneinträge aus Abbildung 3.3 verdeutlicht werden.

3.2.4.1 Klartextversion der Crossreferenzliste für den Master (.pxref)

Tabelle 3.1 zeigt noch einmal die Einträge der Crossreferenzliste zu Abbildung 3.3. Der Vektor V ist ein aufgesplitteter Vektor: Strand 3 befindet sich in allen 3 Teilmodellen, die Strands 0 bis 2 aber nur in den Teilmodellen 0 bis 2. In einem solchen Fall erscheint jeder Vektorstrand als separates Netz (N) in der Liste. Die erste Ziffer nach dem Facility-Namen (V) und der Strand-Nummer (0 ... 3) gibt die Anzahl der Teilmodelle an, auf denen sich die Facility befindet. Danach folgen lediglich die Nummern dieser Teilmodelle.

Eintrag
@N V(0) 2 1 2
@N V(1) 2 1 2
@N V(2) 2 1 2
@N V(3) 3 1 2 3

Tabelle 3.1: Crossreferenzliste zu Abbildung 3.3

Das Format der Einträge für einfache Netze, ungesplittete Vektoren und Arrays ist in Anhang A beschrieben.

3.2.4.2 Klartextversion der Signalschnittlisten für die Slaves (.ext)

Die Tabelle 3.2 zeigt die Signalschnittlisteneinträge des geschnittenen Vektors V. Der erste Eintrag gibt an, ob es sich um eine eingangsseitig (I) oder ausgangsseitig geschnittene Facility (O) handelt. Danach folgt der Facility-Name des Vektors und dessen Länge (4 Strands). Da stets alle Strands übertragen werden, ist es unerheblich, welche Strands tatsächlich angeschlossen sind. Im Falle einer eingangsseitig geschnittenen Facility folgen nun die Anzahl und die Nummern der Ursprungsteilmodelle, für eine ausgangsseitig geschnittene Facility entsprechend die Anzahl und Nummern der Zielteilmodelle.

Eintrag
Signalschnittliste 1:
@I V(4) 1 2
Signalschnittliste 2:
@O V(4) 2 1 3
Signalschnittliste 3:
@I V(4) 1 2

Tabelle 3.2: Signalschnittlisten zu Abbildung 3.3

3.2.4.3 Binärversion der Crossreferenzliste für den Master (.pmod)

Diese Liste enthält die Verteilung der Facilities auf die einzelnen Teilmodelle in kompakter Binärform. Es wird hierbei zwischen *globalen* und *parallelen* Facilities unterschieden.
Eine globale Facility entspricht der Sichtweise des sequentiellen Simulators auf eine Facility: Hier steht der Facility-Name, die Länge der Facility und die Zeilenanzahl (siehe MVLSIM-Manual [Bergman et al. 1999]). Jeder globalen Facility ist in der Regel (d.h. im Falle von einfachen Netzen, ungesplitteten Vektoren und Arrays) eine parallele Facility zugeordnet. Bei aufgesplitteten Vektoren wurden die einzelnen Strands im Zuge der Partitionierung unterschiedlichen Teilmodellen zugeordnet. In diesem Fall existiert für jeden Vektorstrand eine separate parallele Facility.
Die parallelen Facilities enthalten als wichtigste Information die Nummern der Teilmodelle, auf denen sich die Facility befindet. Eine Facility existiert in mehreren Teilmodellen, falls sie eine geschnittene Facility ist oder sie sich im Überlappungsgebiet von Cones befindet, die verschiedenen Teilmodellen zugeordnet wurden (Logikreplikation).
Falls der Master beim Start keine binäre Crossreferenzliste vorfindet, so wird er diese defaultmäßig aus der Klartextliste erzeugen. Damit können nachfolgende parallele Simulationsläufe mit dem gleichen Modell dann bedeutend schneller gestartet werden. Es werden auch komprimierte (gzip) Crossreferenzlisten unterstützt. Diese haben die Endung .pmod.gz .

3.2.4.4 Binärversion der Signalschnittlisten ohne Simulatorindizes (.rmod)

Diese Version der binären Signalschnittliste wird von der Partitionierungsumgebung (*filegen*) erzeugt. Die geschnittenen Facilities werden darin anhand

ihrer Namen identifiziert. Zum schnellstmöglichen Einlesen der Signalschnittliste sind Facility-Indizes allerdings besser geeignet als Namen. Die Facility-Indizes zu einem Facilitity-Namen kann der Partitionierer aber nicht ermitteln - dies kann nur der Simulator selbst tun. Daher war es bisher nötig, im letzten Schritt der Partitionierung das Programm *mvlr2p* zu starten. Mvlr2p ist ein von R. Reilein [Reilein 1998] entwickelter MVLSIM-Client, der mit Hilfe des entsprechenden Teilmodells und des .rmod-Files die Simulatorindizes zu den Facility-Namen ermittelt hat. Das Ergebnis war eine Binärversion der Signalschnittliste mit Simulatorindizes. Diese konnte dann vom parallelen Simulator zu Beginn eingelesen werden. Bei diesem Vorgehen wird MVLSIM aber bereits während der Partitionierung im Vorfeld der eigentlichen Simulation verwendet, wodurch die klare Trennung dieser beiden Schritte verschwimmt. Daher wurde im Rahmen der Anpassung der Listenfiles der dlbSIM um die Fähigkeit des Lesens von .rmod-Files erweitert, so daß die Nachbearbeitung der Signalschnittlisten durch mvlr2p nicht mehr notwendig ist.

3.2.4.5 Binärversion der Signalschnittlisten mit Simulatorindizes (.pmod)

Die .pmod-Signalschnittliste wird in der neuen dlbSIM-Version defaultmäßig beim ersten Simulationslauf mit einem neu partitionierten Modell aus der .rmod-Liste erzeugt. Auf Wunsch kann die .pmod-Liste auch gleich komprimiert werden.
Die .pmod-Signalschnittliste enthält alle Informationen über die geschnittenen Facilities inklusive deren Simulatorindizes. Außerdem existiert aber noch eine weitere Liste mit Simulatorindizes am Ende des .pmod-Files, die nichts mit den geschnittenen Facilities zu tun hat: Diese Liste ist die Symbolliste der Slaves. Sie enthält für jede globale Facility einen Eintrag. Falls diese Facility im Teilmodell vorkommt, steht dort der entsprechende lokale Simulatorindex, andernfalls eine Null[4].

3.2.4.6 Suchreihenfolge beim Lesen der Listenfiles

Der dlbSIM versucht zunächst, die binären Listenfiles einzulesen. Schlägt dies fehl, muß er sich mit den Klartextlisten begnügen, was entsprechend mehr

[4]Bei einem Facility-Zugriff (z.B. Auslesen einer Facility) sendet der Master an den/die betroffenen Slave(s) den globalen Facility-Index. Der Slave verwendet diesen als Offset in seiner Symbolliste und findet dort den passenden lokalen Facility-Index. Dieser lokale Index kann schließlich zum Auslesen der Facility vom zugrundeliegenden MVLSIM verwendet werden.

Zeit in Anspruch nehmen wird. Die konkrete Suchreihenfolge lautet wie folgt:

Crossrefrenzliste (Master):

1. Binärversion (.pmod)
2. Komprimierte Binärversion (.pmod.gz)
3. Klartextversion (.pxref)

Signalschnittlisten (Slaves):

1. Binärversion mit Simulatorindizes (.pmod)
2. Komprimierte Binärversion mit Simulatorindizes (.pmod.gz)
3. Binärversion ohne Simulatorindizes (.rmod)
4. Klartextversion (.ext)

3.2.5 Änderungen an den Listenfiles beim neuen dlbSIM

Die neue Methode zum Austausch der Werte geschnittener Facilities hat auch Änderungen an den Listenfiles nach sich gezogen. Hierzu mußten sowohl die Erzeugung der Listenfiles in *filegen* als auch die Routinen zum Einlesen derselben innerhalb des dlbSIM angepaßt werden. Die Erläuterungen in Anhang A beziehen sich bereits auf das neue Listenfileformat.
An den Signalschnittlisten wurden im Zusammenhang mit der neuen Methode des Austausches der Werte der geschnittenen Facilities folgende Änderungen vorgenommen:

- Da beim Austausch eines geschnittenen Vektorwertes nun stets sämtliche Strands mit einem Zugriff übertragen werden, reicht es aus, wenn dieser Vektor lediglich ein einziges Mal samt zugehöriger Längeninformation in der Liste auftritt. Bisher mußten alle Strands einzeln übertragen werden, so daß für jeden Vektor-Strand ein separater Eintrag notwendig war.

- Falls eine geschnittene Facility das Parallel-Instance-Feature verwendet, so werden alle Instanzen mit einem einzigen *ps_getfacdata()* bzw. *ps_putfacdata()*-Aufruf erfaßt. Daher müssen PIF-Facilities auch nur ein Mal in der Signalschnittliste auftreten. Zuvor mußte jede PIF-Instanz einzeln übertragen werden weshalb ein extra Listeneintrag für jede Instanz erforderlich war.

Darüber hinaus wurden auch kleine Änderungen vorgenommen, die nichts mit dem Austausch der Werte geschnittener Facilities zu tun haben:

- Der Datentyp des *Bitbreite/Bitposition*-Feldes im .pmod-File des Masters bzw. des *Bitbreite*-Feldes der binären Slave-Files wurde von *signed short* auf *signed int* erweitert. Damit werden nun auch Vektoren mit bis zu 65536 Strands unterstützt. Zuvor waren maximal 32768 Strands möglich gewesen.
- Die Interpretationen des ehemaligen *Arraykennung*-Feldes im .pmod-File des Masters (siehe [Reilein 1998], S. 68) wichen im *filegen* und dlbSIM voneinander ab. Dieser Fehler wurde behoben. Der Eintrag trägt jetzt den Namen *Facility-Kennung* (vergleiche Anhang A).

3.2.6 Interne Datenstrukturen der neuen dlbSIM-Slaves zum Austausch der Werte geschnittener Facilities

Ein Slave-Knoten des dlbSIM lädt bei aktivierter dynamischer Lastbalancierung zu Beginn mehrere Teilmodelle in den Speicher, von denen jeweils eine Teilmenge aktiv an der Simulation teilnimmt. Der zugrundeliegende MVLSIM legt pro geladenem Teilmodell eine Simulations-Kontrollstruktur *sc* (sc ist Instanz der Struktur SIM_CB) an, in welcher sämtliche Informationen über das Teilmodell enthalten sind, so auch die Erweiterungsstruktur für den dlbSIM *dlb* (Instanz der Struktur SC_EXT). Die parallele Slave-Komponente *stexsim* besitzt des weiteren eine knotenbezogene Datenstruktur *nc* (struct NODE_CB), in welcher diejenigen Informationen abgelegt werden, die einen Knoten als Ganzes betreffen.

3.2.6.1 Teilmodellbezogene Datenstrukturen *COMFAC* und *PARFAC*

Nach erfolgreichem Laden der Teilmodelle wird ein Slave-Knoten für jedes Teilmodell die zugehörige Signalschnittliste einlesen. Die binäre .pmod-Liste enthält zwei Datenbereiche, die nahezu unverändert als Datenstrukturen zum Austausch der geschnittenen Facility-Werte in den Speicher übernommen werden.
Die erste Datenstruktur ist die in Abbildung 3.4 dargestellte *COMFAC*-Struktur (Communication-Facility-Struktur).

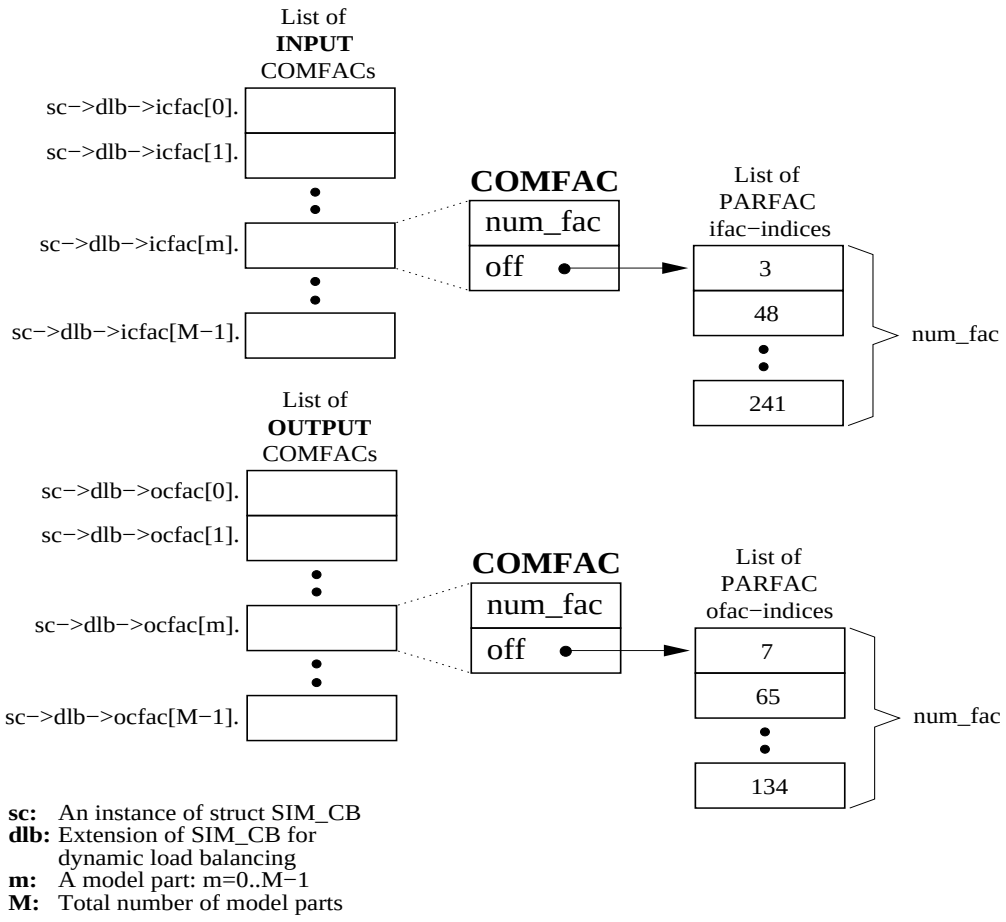


Abbildung 3.4: Teilmodellbezogene Slave-COMFAC-Struktur

Es gibt zwei verschiedene Instanzen der COMFAC-Struktur: Die Input-Communication-Facilities (icfac's) für die eingangsseitig geschnittenen Facilities und die Output-Communication-Facilities (ocfac's) dementsprechend für die ausgangsseitig geschnittenen Facilities eines Teilmodells. Eine Input-Communication-Facility speichert Informationen über alle eingangsseitig geschnittenen Facilities, die von Teilmodell m in das eigene Modell führen. Daher gibt es genau so viele Input-Communication-Facilities wie Teilmodelle. Der erste Eintrag (num_fac) gibt die Anzahl der Input-Facilities an, die von Teilmodell m kommen. Der zweite Eintrag (off) ist ein Zeiger auf eine Liste sogenannter $PARFAC$-Indizes. Für jede eingangsseitig geschnittene Facility existiert eine eigene PARFAC-Struktur, die nachfolgend beschrieben wird und nähere Informationen über die entsprechende Facility enthält. Alle Input-Communication-Facilities sind in einer Liste zusammengefaßt und sind über die dlbSIM-Erweiterungsstruktur $sc \rightarrow dlb$ erreichbar. Der Listenindex m bezeichnet dann gerade die Nummer desjenigen

Teilmodells, von dem die eingangsseitig geschnittenen Facilities ausgehen. Für die COMFAC-Struktur des eigenen Teilmodells ist *num_fac*=0 und es gibt auch keine PARFAC-Indizes.
Analoges gilt für die Output-Communication-Facilities, welche Informationen über diejenigen ausgangsseitig geschnittenen Facilities speichern, die vom eigenen Teilmodell zum Teilmodell *m* führen.

Die *PARFAC*-Strukturen (Parallel-Facility-Strukturen) sind ebenfalls Bestandteil der binären .pmod-Listenfiles. Abbildung 3.5 zeigt deren Einbindung in die *sc*→*dlb*-Struktur eines Teilmodells.

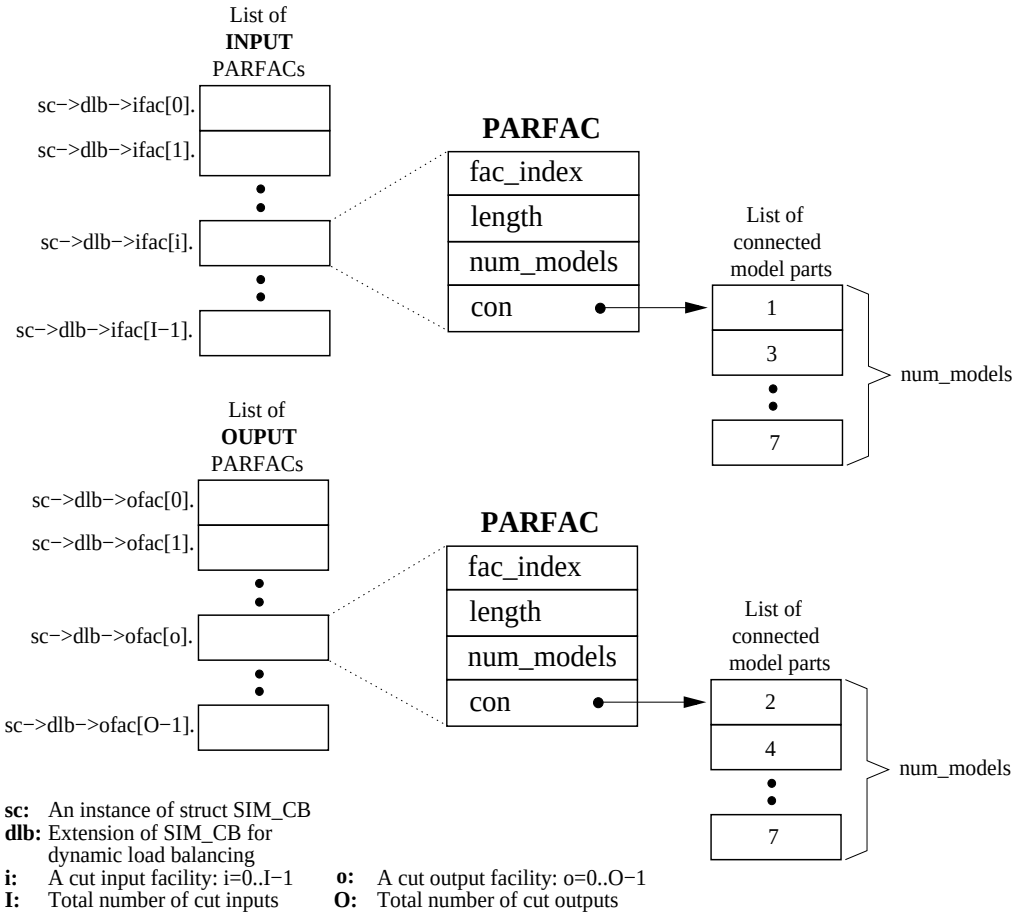


Abbildung 3.5: Teilmodellbezogene Slave-PARFAC-Struktur

Die *ifac*-Instanzen (Input-Facilities) speichern sämtliche Informationen zu den eingangsseitig geschnittenen Facilities eines Teilmodells, die *ofac*-Instanzen (Output-Facilities) enhalten die Daten zu den ausgangsseitig geschnittenen Facilities. Die Input- und Output-Facilities sind jeweils wiederum

in einer Liste zusammengefaßt. Der Listenindex (PARFAC-Index) bezeichnet die Nummer der geschnittenen Facility. Die PARFAC-Struktur selbst beinhaltet zunächst den lokalen Facility-Index (*fac_index*) sowie die Bitbreite (*length*) der geschnittenen Facility. Der Eintrag *num_models* gibt im Fall einer Input-Facility die Anzahl der Teilmodelle an, die diese Facility speisen. Für eine Output-Facility ist es entsprechend die Anzahl der Teilmodelle, die von dieser Facility gespeist werden. Zu guter Letzt ist *con* ein Zeiger auf ein Feld mit *num_models* Einträgen, welches die Nummern der jeweiligen Teilmodelle enthält.

3.2.6.2 Knotenbezogene *NODE_COMM_DAT*-Struktur

Der Austausch der geschnittenen Facility-Werte im Transfer-Schritt des parallelen Clock-Cycle-Algorithmus erfolgt knotenbezogen: Ein Slave-Knoten tauscht hierbei die geschnittenen Facility-Werte all seiner aktiven Teilmodelle mit den entsprechenden anderen Slaves aus. Die Informationen über die geschnittenen Facilities in den COMFAC-Strukturen sind jedoch immer nur auf ein Teilmodell bezogen. Zwar ließe sich der Facility-Austausch auch allein über die COMFAC-Strukturen realisieren, allerdings würde es dann aufgrund des umständlichen Zusammensammelns der zu einem Knoten passenden COMFAC's zu erheblichen Leistungseinbußen kommen.
Aus diesem Grund wurden mit dem Erscheinen des dlbSIM die sogenannten *NODE_COMM_DAT*-Strukturen (Node-Communication-Data) eingeführt, welche eine knotenweise Gruppierung der auszutauschenden Facilities realisieren. Mit dem neuen dlbSIM haben sich auch an dieser Struktur Änderungen ergeben, so daß sie im folgenden noch einmal vollständig beschrieben werden soll. In Abbildung 3.6 ist der Aufbau der neuen NODE_COMM_DAT-Struktur dargestellt.

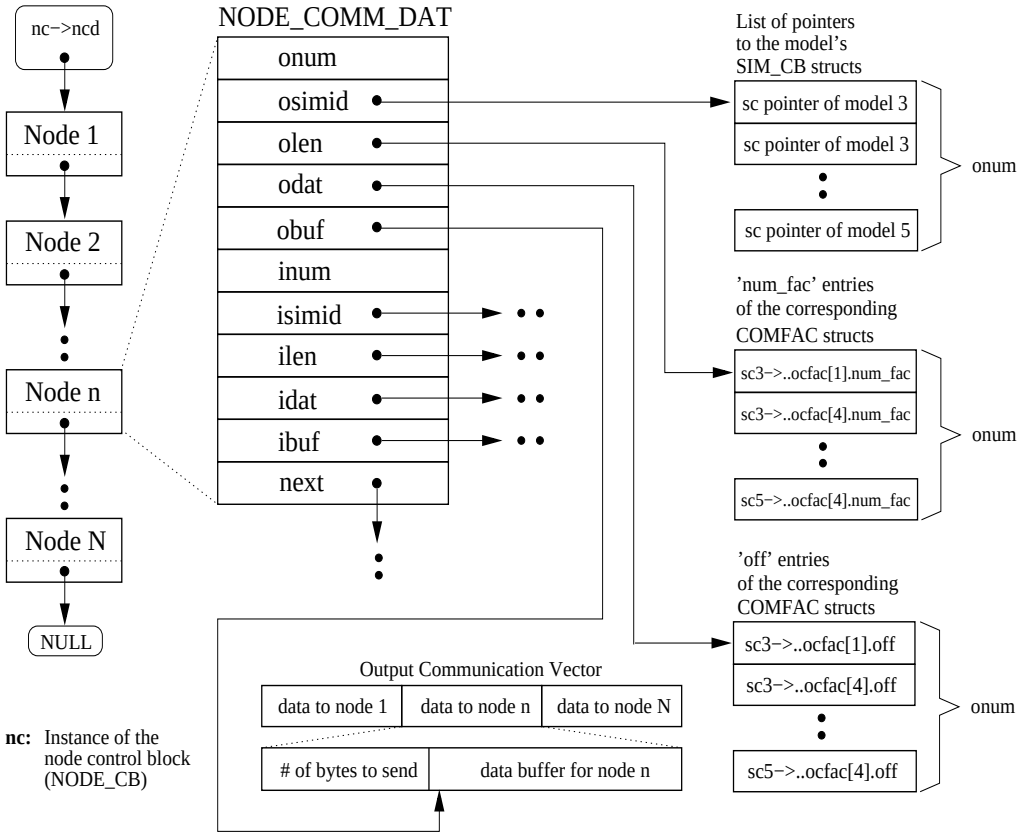


Abbildung 3.6: Knotenbezogene NODE_COMM_DAT-Struktur

Die knotenbezogene *nc*-Struktur enthält einen Zeiger auf eine einfach verkettete Liste von NODE_COMM_DAT-Strukturen. Da der Facility-Austausch in der Regel mit allen Slave-Knoten erfolgt, hat diese Liste genau so viele Einträge wie es Slave-Knoten gibt[5]. Die n-te NODE_COMM_DAT-Struktur enthält demzufolge alle Informationen über die geschnittenen Facilities, die der eigene Knoten mit dem Knoten n auszutauschen hat.
In die NODE_COMM_DAT-Strukturen gehen nur diejenigen geschnittenen Facilities ein, welche sich in aktiven Teilmodellen befinden. Daher ist es notwendig, die Strukturen nach jeder Modellverschiebung wieder neu aufzubauen. Dies erfolgt in der Funktion *s_create_ncd()*.
Beim Aufbau der zu einem Knoten gehörenden NODE_COMM_DAT-Struktur wird wieder zwischen ein- und ausgangsseitig geschnittenen Facilities unterschieden. Die nachfolgenden Erläuterungen zu den Einträgen der

[5] Auch für den eigenen Knoten existiert eine NODE_COMM_DAT-Struktur, da zwischen dessen aktiven Teilmodellen ebenfalls geschnittene Facilities existieren können. Allerdings werden die Facility-Daten in diesem Fall nicht in den Kommunikationsvektor geschrieben, sondern lokal kopiert.

ausgangsseitig geschnittenen Facilities gelten aber analog auch für die eingangsseitigen Facilities:
Der erste Eintrag, *onum*, gibt die Gesamtanzahl der Modell-zu-Modell-Verbindungen an, die von aktiven Modellen des eigenen Knotens ausgehen und zu aktiven Modellen des Knotens n führen. Zur Illustration ist in Algorithmus 3 die Ermittlung von *onum* dargestellt:

Algorithmus 3 Bestimmung von *ncd*→*onum* für Knoten n

```
/* Dieser Algorithmus ist Teil von s_create_ncd(). */
/* Er ermittelt den Eintrag ncd→onum für die zu */
/* Knoten n gehörige NODE_COMM_DAT-Struktur. */
onum = 0
for i = 0 to num_of_active_local_models − 1 do
  for j = 0 to total_num_of_models − 1 do
    if model j is loaded on node n then
      /* Gibt es ausgangsseitig geschnittene Facilities, */
      /* die von Modell i zu Modell j führen ? */
      if sc(of model i)→dlb→ocfac[j].num ≠ 0 then
        onum++
      end if
    end if
  end for
end for
```

Das Feld *osimid* ist ein Zeiger auf eine Liste mit Zeigern auf die SIM_CB-Strukturen derjenigen aktiven Teilmodelle des eigenen Knotens, die eine ausgangsseitige Modell-zu-Modell-Verbindung zu einem Modell des Knotens n besitzen.
Die Felder *olen* und *odat* enthalten für jede dieser Modell-zu-Modell-Verbindungen die *num_fac*- und *off*-Einträge der zugehörigen COMFAC-Strukturen. Im Beispiel der Abbildung 3.6 befinden sich auf dem eigenen Knoten u.a. die aktiven Modelle Nr. 3 und 5. Beide besitzen ausgangsseitig geschnittene Facilities, die zu aktiven Modellen des Knotens n führen. Modell 3 besitzt ausgangsseitige Verbindungen zu den Modellen 1 und 4 des Knotens n, währenddessen Modell 5 dort lediglich mit Knoten 4 in Verbindung steht. Der letzte Eintrag der ausgangsseitigen Gruppe ist *obuf*. Er ist ein Zeiger auf denjenigen Datenbereich des Output-Kommunikationsvektors, der zu Knoten n übertragen wird. Wie man aus Abbildung 3.6 ersehen kann, ist der Kommunikationsvektor ein zusammenhängender Speicherbereich, der nacheinander die Daten für die einzelnen Knoten enthält. Am Anfang des Datenfeldes steht

die Länge des nachfolgenden Nutzdatenbereiches für die geschnittenen Facilities, gefolgt von den Facility-Werten selbst. Die korrekten Längeninformationen werden bereits beim Aufbau der NODE_COMM_DAT-Strukturen in die Kommunikationsvektoren eingetragen. Die Ermittlung der Längeninformationen erfolgt durch Addition der *length*-Werte der geschnittenen Facilities aus den PARFAC-Strukturen. Vor dem Aufbau der NODE_COMM_DAT-Strukturen werden die Teilmodelle nach ihren Nummern sortiert. Damit wird sichergestellt, daß die sendenden und empfangenden Knoten beim Austausch des Kommunikationsvektors die gleiche Reihenfolge bezüglich der Teilmodelle einhalten.

3.2.7 Paralleler Clock-Cycle-Algorithmus des neuen dlbSIM

Der parallele Clock-Cycle-Algorithmus der Simulatoren parallelTEXSIM bzw. parallelMVLSIM wurde bereits kurz in Kapitel 2 im Rahmen der parallelen Modellsimulation vorgestellt. Beim dlbSIM wird dieser Algorithmus prinzipiell in der gleichen Weise abgearbeitet, allerdings nur auf denjenigen Slave-Knoten, die über mindestens ein aktives Teilmodell verfügen. Dies wird erreicht, indem der Master den Befehl zur Simulation nur an aktive Slave-Knoten verschickt.
Zu Beginn des parallelen Clock-Cycle-Algorithmus wird zunächst geprüft, ob die in Abbildung 3.6 dargestellte Liste von NODE_COMM_DAT-Strukturen neu aufzubauen ist. Dies ist einmal zu Beginn der Simulation und dann nach jeder Modellverschiebung erforderlich. Danach können die einzelnen Schritte des parallelen Clock-Cycles abgearbeitet werden.

3.2.7.1 CLOCK

CLOCK evaluiert nacheinander alle aktiven Teilmodelle eines Slave-Knotens. Dazu wird das entsprechende Teilmodell zunächst über die MVLSIM-Funktion *txsetid()* selektiert. Der anschließende *clock(1)*-Aufruf bezieht sich dann auf das selektierte Teilmodell und simuliert dort einen Clock-Zyklus.

3.2.7.2 GET

GET liest nun unter Verwendung der NODE_COMM_DAT-Strukturen die Werte der ausgangsseitig geschnittenen Facilities aus den aktiven Teilmodellen aus und schreibt diese in den Output-Kommunikations-Vektor. Das Auslesen der Facility-Werte aus den Modellen erfolgt im Gegensatz

zum originalen dlbSIM jetzt über die PS-API-Funktion *ps_getfacdata()*. Algorithmus 4 zeigt die Funktionsweise des GET-Schrittes im Detail:

Algorithmus 4 GET-Schritt des parallelen Clock-Cycle-Algorithmus

```
ncd = nc→ncd                          {ncd zeigt nun auf die erste NCD-Struktur}
for node = 1 to number_of_slave_nodes do
  if node ≠ this_node then
    obuf = ncd.obuf   {Zeiger auf Datenbereich des Kommunik.-vektors}
    for i = 0 to ncd→onum − 1 do
      o_sc = ncd→osimid[i]   {Zeiger auf SIM_CB des Output-Modells}
      num_fac = ncd→olen[i]  {Anzahl der geschnitt. Output-Facilities}
      off = ncd→odat[i]          {Zeiger auf Liste mit PARFAC-Indizes}
      ofac = o_sc→dlb→ofac     {Zeiger auf Output-PARFAC-Struktur}
      txsetid(o_sc)                    {Aktivierung des Output-Modells}
      for j = 0 to num_fac − 1 do
        facidx = ofac[*off].fac_index  {Lokaler Index der j-ten Facility}
        length = ofac[*off].length              {Länge der j-ten Facility}
        incr = ps_getfacdata(obuf, facidx, length)
        obuf = obuf + incr    {Zeiger im Kommunik.-verktor erhöhen}
        off++          {off zeigt jetzt auf den nächsten PARFAC-Index}
      end for
    end for
  end if
  ncd = ncd.next {NODE_COMM_DAT-Struktur des nächsten Knotens}
end for
```

Falls geschnittene Facilities zwischen aktiven Teilmodellen des eigenen Knotens bestehen, so werden diese ebenfalls über die NODE_COMM_DAT-Strukturen referenziert. Allerdings werden die Facility-Werte in diesem Falle nicht in den Kommunikationsvektor geschrieben, sondern lokal kopiert. Somit kann die Zeit für das Schreiben und spätere Lesen des Kommunikationsvektors eingespart werden.

3.2.7.3 TRANSFER

In TRANSFER-Schritt werden die Daten der Kommunikationsvektoren zwischen allen Slave-Knoten ausgetauscht. Hierzu wird die von J. Löser entwickelte Funktion *mpc_index3()* verwendet. Diese Funktion basiert auf den zwei MPL-Funktionen *mpc_send()* und *mpc_receive()*, welche einen nichtblockierenden Punkt-zu-Punkt Datentransfer realisieren. Der Pseudocode von *mpc_index3()* ist in [Löser 1998], S. 50 zu finden.

3.2.7.4 PUT

PUT arbeitet wie GET ebenfalls über den NODE_COMM_DAT-Strukturen. Es liest die neuen Werte der eingangsseitig geschnittenen Facilities aus dem Input-Kommunikations-Vektor aus und schreibt diese mit Hilfe von *ps_putfacdata()* in die entsprechenden Teilmodelle zurück.

3.3 Experimente zum neuen dlbSIM mit deaktivierter Lastbalancierung

Die nachfolgenden Experimente untersuchen den Speedup des neuen parallelen Simulators im Vergleich zum sequentiellen MVLSIM unter Verwendung einer variierenden Anzahl von Slave-Knoten. Es wird zudem auch untersucht, inwieweit sich unterschiedliche Kommunikationsmedien (HighPerformance-Switch/Ethernet) auf die parallele Simulationszeit auswirken.

3.3.1 Testumgebung

Zur Durchführung der Experimente konnte eine IBM *SP2 (Scalable POWERparallel System 2)* mit acht Prozessorknoten sowie einem *HighPerformanceSwitch* genutzt werden. Dieser ermöglicht sogenannte *Any-to-Any*-Verbindungen zwischen allen Prozessorknoten. Die maximale Bandbreite beträgt pro Verbindung 40 MB/s. Näheres zur Architektur einer *SP2* kann in [Hennings 1998] nachgelesen werden. Zusätzlich sind die Knoten auch über 10 MBit Ethernet verbunden. Alle Knoten der *SP2* sind in ihrer Hardware identisch und lassen sich wie folgt charakterisieren:

- 132 MHz *Power2* SingleChip CPU
- 32 KB L1 Befehlscache
- 128 KB L1 Datencache
- kein L2 Cache
- 1 GB Hauptspeicher
- 2 GB virtueller Speicher

Softwareseitig stand auf allen Knoten das Betriebssystem *AIX 4.2* sowie das *IBM Parallel Environment* zur Verfügung.

Nachfolgende Bedingungen blieben in allen Experimenten unverändert:

- Es wurden stets 5000 Clock-Zyklen simuliert. Die Erzeugung von Trace- und Logfiles wurde deaktiviert.
- Die Partitionierung wurde unter Verwendung des *PEnv* und der evolutionären Algorithmen durchgeführt. Für die Erzeugung der Vorpartition kam *STEP* zum Einsatz.
- Alle erzeugten Modelle verwenden 2-wertige Logik und wurden nicht optimiert.
- Die Erzeugung störender, fremder CPU- und Netzlast konnte für den Zeitraum der Experimente weitesgehend ausgeschaltet werden.

3.3.2 Durchführung der Experimente

Die Experimente wurden mit insgesamt drei verschiedenen Protos durchgeführt: Zunächst mit dem relativ kleinen Proto des **PowerPC 601**, welches knapp 50500 Logikbausteine enthält. Der Großteil der Untersuchungen wurde jedoch mit dem Proto eines Prozessorkerns des *Power4* GigaHertz-Prozessors **GP_Core** (918000 Logikbausteine) sowie mit dem großen Proto des *S/390* Prozessors **Monet** (2.7 Millionen Logikbausteine) durchgeführt. Mit letzterem Proto hatte bereits D. Döhler [Döhler 1996] im IBM Entwicklungslabor Böblingen auf einer *SP2* Messungen zum parallelTEXSIM durchgeführt, so daß sich hier Vergleichsmöglichkeiten ergeben.
Zunächst wurde aus jedem Proto ein vollständiges, unpartitioniertes Modell gebaut und mit dem MVLSIM auf einem Knoten der *SP2* simuliert, um die sequentielle Simulationszeit für dieses Modell zu ermitteln. Anschließend wurden die Protos in drei bis fünf Blöcke partitioniert und damit die Teilmodelle und Listenfiles für die parallele Simulation erzeugt. Während der Simulation wurde auf jeden Knoten nur ein Teilmodell geladen, da bei ausgeschalteter Lastbalancierung keine redundante Verteilung der Teilmodelle auf die Slave-Knoten vorgenommen wird (analog zum parallelMVLSIM). Auch für den Master wurde ein eigener Knoten reserviert. Im Laufe der parallelen Simulation wurde für jeden Slave-Knoten zunächst die totale Simulationszeit (*Total run time*) gemessen. Diese setzt sich aus der Zeit für die Initialisierung (Laden des Teilmodells und der Listenfiles) sowie folgenden drei Zeiten zusammen:

- **Clocking:** Das ist diejenige Zeit, die für die eigentliche Simulation verwendet wurde. Idealerweise sollte sie den größten Teil der Gesamtzeit ausmachen und bei guter Partitionierung auf allen Knoten annähernd

gleich groß sein. Der sequentielle MVLSIM verbringt in diesem Experiment fast seine gesamte Laufzeit mit dem Clocken. (Ein geringer Zeitanteil wird für die Initialierung und das Laden des Modells benötigt.)

- **Get/Put:** Bezeichnet die Zeit, welche insgesamt für das Auslesen und Zurückschreiben der Werte der geschnittenen Facilities in den Funktionen *ps_getfacdata()* bzw. *ps_putfacdata()* verbracht wurde.

- **Transfer:** Das ist die Gesamtzeit, die für den Austausch der geschnittenen Facility-Werte zwischen den Knoten verwendet wurde. Diese Zeit beinhaltet auch mögliche Wartezeiten auf den „langsamsten Knoten", da ein Datenaustausch erst dann erfolgen kann, wenn alle Knoten zur Kommunikation bereit sind. Implementiert ist der Datenaustausch über nicht blockierende *mpc_send()* und *mpc_receive()* Aufrufe. Mittels *mpc_wait()* wird das Ende der Kommunikation abgewartet.

Für jede der soeben genannten Zeiten wurde außerdem noch deren genaue Zusammensetzung (USR-, SYS- und REAL-Time) bestimmt. Dies erfolgte hauptsächlich mit dem Ziel, mögliche Fremdlasteinwirkungen durch andere Nutzer zu erkennen.

- **USR:** Das ist die CPU-Zeit, die ein Prozeß für die Abarbeitung von Befehlen im User-Mode verbraucht hat.

- **SYS:** Diese CPU-Zeit hat ein Prozeß in Betriebssystemfunktionsaufrufen verbracht. Sowohl die USR- als auch die SYS-Time wurden mit Hilfe des Funktionsaufrufes *times()* ermittelt.

- **REAL:** Dies ist die tatsächlich benötigte Laufzeit. Sie wurde über *read_real_time()* und *time_base_to_time()* bestimmt. Falls gilt: USR+SYS$\ll$REAL, so deutet dies auf eine Fremdlastsituation hin.

Hinweis: Die ermittelte REAL-Time ist stets exakt. Jedoch sind die von *times()* zurückgelieferten USR- und SYS-Zeiten recht fehlerbehaftet, so daß im Ausnahmefall USR+SYS>REAL werden kann. Dies ist dann auf Meßfehler zurückzuführen.

Schließlich gehen noch folgende Meßgrößen in die Experimente ein, die ebenfalls mit erwähnt werden sollen:

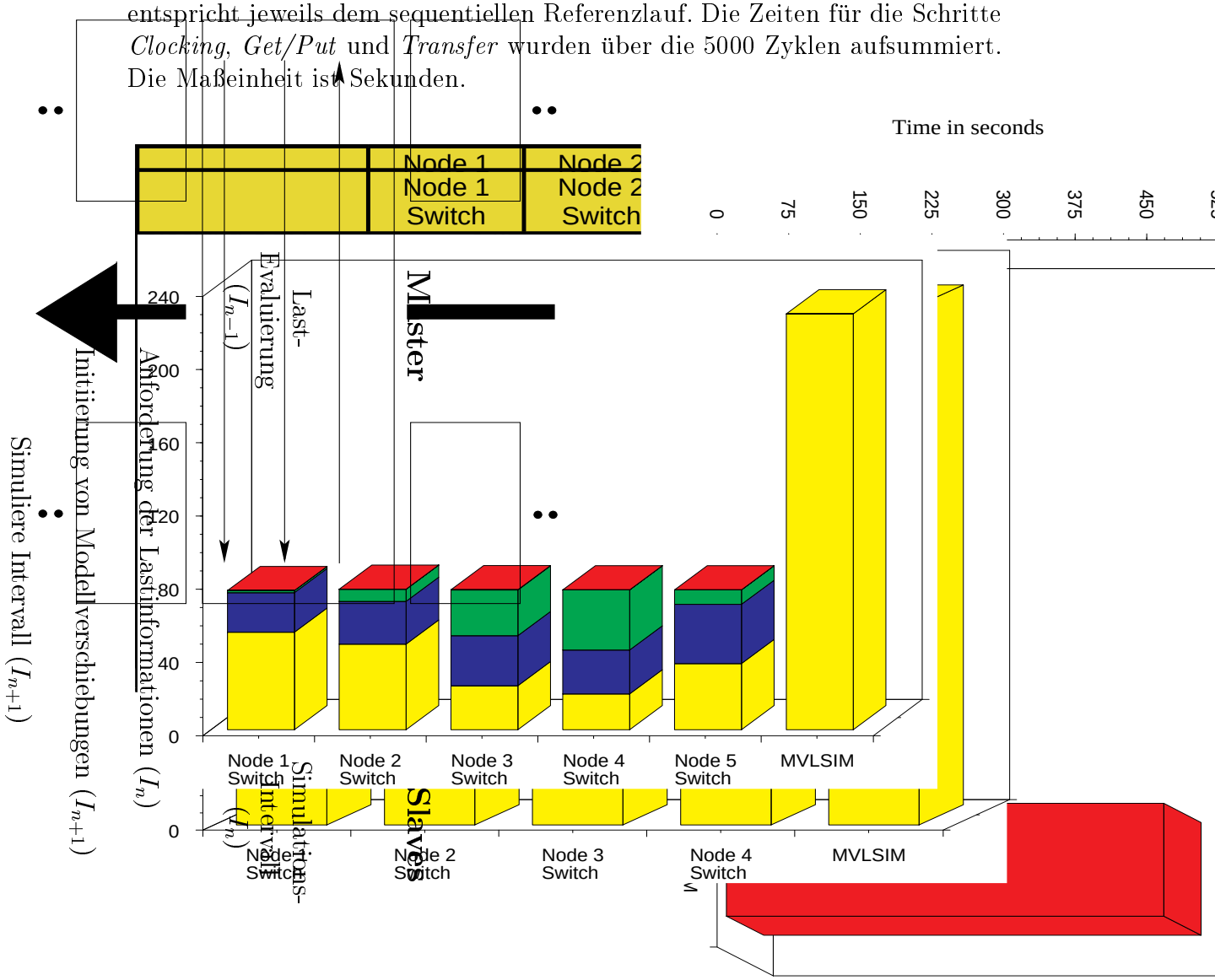
- **GETs per cycle / PUTs per cycle:** Gibt die Anzahl der *ps_getfacdata()* bzw. *ps_putfacdata()* Aufrufe pro Zyklus an. Hiermit läßt sich recht gut der Aufwand für den Austausch der Werte der geschnittenen Facilities abschätzen. Da keine Modellverschiebungen stattfinden, bleiben diese Größen während der gesamten Simulation konstant.
- **Load:** Gibt die exponentiell über die letzte Minute gemittelte und mit 256 multiplizierte Systemlast an. Der Wert wurde mit Hilfe der *rstat()*-Funktion ermittelt.
- **Total run time:** Gibt die insgesamt, für den gesamten Simulationslauf benötige Zeit an. Sie wurde mit einer Stoppuhr gemessen.
- **Speedup:** Das ist der Speedup des parallelen Simulationslaufes im Vergleich zum sequentiellen MVLSIM. Er basiert auf der *Total run time*.

3.3.3 Ergebnisse

PowerPC 601
Aufgrund der geringen Größe des PowerPC 601 hat sich herausgestellt, daß der Overhead der parallelen Simulation (Schritte *Get/Put* und *Transfer*) größer wurde als die Zeit für die eigentliche Simulation (Schritt *Clocking*). Somit benötigte ein paralleler Simulationslauf mit drei Slave-Knoten letztendlich mehr Zeit als ein sequentieller Lauf mit dem MVLSIM. Daran läßt sich erkennen, daß ein sinnvoller Einsatz des parallelen Simulators nur mit größeren Modellen möglich ist.

GP_Core

Das Proto GP_Core wurde drei mal partitioniert, in Partitionen zu drei, vier und fünf Blöcken. Sämtliche Partitionen wurden mit Switch-Kommunikation simuliert. Zusätzlich wurden die Vierer- und Fünfer-Partitionen auch mittels Ethernet-Kommunikation untersucht. Die Abbildungen 3.3, 3.4 und 3.5 zeigen sämtliche Meßwerte in Tabellenform. Die mit MVLSIM betitelte Spalte entspricht jeweils dem sequentiellen Referenzlauf. Die Zeiten für die Schritte *Clocking*, *Get/Put* und *Transfer* wurden über die 5000 Zyklen aufsummiert. Die Maßeinheit ist Sekunden.



4.1.2 Evaluierung der Lastinformationen der Slaves

Die Erfassung und Auswertung von Lastinformationen erfolgt mit dem Ziel der Erkennung und Beseitigung von Fremdlasteinflüssen. Aus den gesammelten Lastdaten ermittelt der Master für jeden Slave-Knoten zunächst einen sogenannten *Lastindex*. Mit dessen Hilfe kann der rekursive Lastbalancierungsalgorithmus eine *Zykluszeitvorhersage* zur Evaluierung von testweise erzeugten, neuen Modellverteilungen durchführen.

4.1.2.1 Lastindexbestimmung

Der Lastindex soll Auskunft über die applikationsbezogene Rechengeschwindigkeit eines Knotens geben. Hierbei soll ein Programm auf einem Knoten mit Lastindex 2 doppelt soviel Zeit zur Berechnung einer Aufgabe benötigen als ein Programm auf einem Knoten mit Lastindex 1.
Bei der Ermittlung des Lastindexes muß zwischen Knoten mit aktiven Teilmodellen und Knoten ohne aktive Teilmodelle unterschieden werden:

- Auf Knoten mit aktiven Teilmodellen dient die benötigte Zeit zum Auslesen und Zurückschreiben der Werte der geschnittenen Facilities als Basis zur Lastindexberechnung. Diese Zeit wächst linear mit der bekannten Zahl der geschnittenen Facilities und hat einen engen Bezug zur tatsächlichen Ausführungszeit eines Slave-Prozesses.

- Bei Knoten ohne aktive Teilmodelle werden keine geschnittenen Facilites ausgelesen oder zurückschrieben. Daher steht diese Zeit hier nicht zur Verfügung. Für die Lastindexberechnung kann demzufolge lediglich die vom Betriebssystem ermittelte *Load* des Knotens herangezogen werden. Allerdings ist die Load-Zahl eines Knotens nicht direkt mit der Zeit zum Auslesen und Zurückschreiben der Werte der geschnittenen Facilities vergleichbar, da dies zwei verschiedene Wertungssysteme sind. Um dennoch eine Vergleichsbasis für die beiden unterschiedlichen Lastindexverfahren zu erhalten, müssen auch auf den Knoten mit aktiven Teilmodellen die Load-Zahlen ermittelt werden. Anschließend bildet man dort ein Verhältnis aus den Zeiten zum Auslesen/Rückschreiben der geschnittenen Facilities und den Load-Zahlen. Mit Hilfe dieses Verhältnisses kann nun auf einem Knoten ohne aktive Teilmodelle aus der dort vorliegenden Load-Zahl diejenige Zeit abgeschätzt werden, die der Knoten zum Auslesen/Rückschreiben des Wertes einer geschnittenen Facility benötigen würde, wenn er ein aktives Teilmodell hätte.

In [Löser 1998] sind sämtliche Details der Lastindexbestimmung zu finden, allerdings leider ohne Angabe der konkreten Berechnungsvorschrift. Diese soll nun nachgereicht werden.
Zur näheren Erläuterung des Berechnungsalgorithmus ist es zunächst erforderlich, die folgenden Symbole zu definieren:

- $\mathbf{K}$: Menge alle Slave-Knoten.
- $\mathbf{K_A}$: Menge aller Slave-Knoten, die momentan mindestens ein aktives Teilmodell besitzen.
- $\mathbf{K_I}$: Menge aller Slave-Knoten, die momentan kein aktives Teilmodell besitzen. ($K_A \cup K_I = K$ und $K_A \cap K_I = \emptyset$)
- $\mathbf{M}$: Menge aller Teilmodelle.
- $\mathbf{M_A(k)},\ \mathbf{k} \in \mathbf{K_A}$: Menge aller aktiven Teilmodelle des Knotens k.
- $\mathbf{i, j} \in \mathbf{M},\ \mathbf{i} \neq \mathbf{j}$: zwei verschiedene Teilmodelle aus M.
- $\mathbf{n(i, j)}$: Anzahl der geschnittenen Facilities, die zwischen den Modellen i und j auszutauschen sind.

Berechnungen auf Knoten mit aktiven Teilmodellen ($k \in K_A$):
Man ermittle zunächst die Anzahl der auf dem Knoten k insgesamt auszutauschenden geschnittenen Facilties $\mathbf{n(k)}$:

$$n(k) = \sum_{\substack{i \in M_A(k) \\ j \in M_A(l),\ l \in K_A,\ l \neq k}} n(i,j)$$

Sei $\mathbf{t(k)}$ die Zeit, welche Knoten k insgesamt für das Auslesen und Zurückschreiben all seiner $n(k)$ geschnittenen Facility-Werte benötigt hat. Dann ergibt sich die Zeit zur Bearbeitung einer Facility $\mathbf{t_F(k)}$ aus:

$$t_F(k) = \frac{t(k)}{n(k)}$$

Die Durchschnittszeit über alle Knoten zur Bearbeitung einer Facility $\mathbf{t_{AVG}}$ läßt sich wie folgt ermitteln:

$$t_{AVG} = \frac{\sum_{k \in K_A} t(k)}{\sum_{k \in K_A} n(k)}$$

Nun kann man bereits den Lastindex $\mathbf{V_k}$ eines Knotens mit aktiven Teilmodellen bestimmen:

$$V_k = \frac{t_F(k)}{t_{AVG}}$$

Jetzt wird das bereits erwähnte Verhältnis aus der Zeit zur Bearbeitung einer Facility auf einem Knoten und der dortigen Load-Zahl gebildet. Hierbei sei $\mathbf{l(k)}$ die ermittelte Load-Zahl[2] des Knotens k. Dann ist $\mathbf{x}$ der lastunabhängige Mittelwert über alle Knoten mit aktiven Teilmodellen für das Auslesen/Rückschreiben des Wertes einer geschnittenen Facility:

$$x = \frac{\sum_{k \in K_A} \frac{t_F(k)}{l(k)}}{\mid K_A \mid}$$

Berechnungen auf Knoten ohne aktive Teilmodelle ($k \in K_I$):
Auf Knoten ohne aktiven Teilmodellen kann nun x verwendet werden, um diejenige Zeit $\mathbf{t^*_F(k)}$ abzuschätzen, die ein Knoten aus K_I für das Auslesen/Rückschreiben des Wertes einer geschnittenen Facility benötigen würde, wenn er ein aktives Teilmodell hätte:

$$t^*_F(k) = x * l(k), \qquad (k \in K_I)$$

Nun ist es möglich, auch für Knoten aus K_I den Lastindex $\mathbf{V_k}$ zu berechnen:

$$V_k = \frac{t^*_F(k)}{t_{AVG}}$$

[2]Die vom Betriebssystem ermittelten Load-Zahlen sind genau genommen nur auf homogenen Rechnerknoten vergleichbar. In einem heterogenen Workstation-Cluster sollten die Load-Zahlen mit einem Faktor multipliziert werden, der die Leistungsfähigkeit eines jeden Knotens bezüglich der betreffenden Applikation beschreibt. Im Experiment wurde ein solcher Faktor für alle Knoten eines stark heterogenen Workstation-Clusters auf der Basis eines sequentiellen MVLSIM-Testlaufes ermittelt. Die Güte der ermittelten Faktoren ließ sich überprüfen, indem während eines dlbSIM-Laufes auf einem Knoten soviel Fremdlast erzeugt wurde, daß dieser keine aktiven Teilmodelle mehr besaß, denn nur in diesem Fall gehen die Load-Zahlen in den Lastindex ein. Die Testergebnisse waren zufriedenstellend, solange keiner der Knoten knapp an Arbeitsspeicher wurde und Speicherbereiche auf Festplatte auslagern mußte.

4.1.2.2 Zykluszeitvorhersage

Der im nachfolgenden Unterkapitel beschriebene Lastbalancierungsalgorithmus wird durch testweises[3] Verschieben von Teilmodellen versuchen, eine neue Verteilung der Teilmodelle zu finden, welche gegenüber der Ausgangsverteilung eine geringere Berechnungszeit benötigt. Um unter der Vielzahl der erzeugten potentiellen Modellverteilungen die günstigste auswählen zu können, muß für jede testweise erzeugte Verteilung die parallele Zykluszeit abgeschätzt werden. Da der langsamste Knoten die erreichbare Simulationsgeschwindigkeit bestimmt, ergibt sich die parallele Zykluszeit aus der Zykluszeit des langsamsten Knotens. Die Zykluszeit eines Knotens läßt sich wiederum aus der Summe der Zykluszeiten all seiner aktiven Teilmodelle ermitteln.
Unter Verwendung der gesammelten Meßwerte (teilmodellbezogene Evaluationszeiten und den Zeiten für das Auslesen und Rückschreiben der geschnittenen Facilities) läßt sich für alle aktiven Teilmodelle deren Zykluszeit bezüglich der Ausgangsverteilung berechnen. Diese Modell-Zykluszeiten sind lediglich für denjenigen Knoten gültig, auf dem das entsprechende Teilmodell in der Ausgsangsverteilung aktiv ist. Dividiert man nun diese knotenbezogenen Modell-Zykluszeiten durch den Lastindex des zugehörigen Knotens, erhält man eine knotenunabhängige, gewichtete Modell-Zykluszeit. Zur Vorhersage der Zykluszeit eines Modells nach dessen testweiser Verschiebung auf einen anderen Knoten genügt es nun, die gewichtete Modell-Zykluszeit mit dem Lastindex des Zielknotens zu multiplizieren. Nachfolgend sind noch einmal alle Schritte der Zykluszeitvorhersage im Detail aufgeführt:

Für die Vorhersage der parallelen Zykluszeit ist es zunächst erforderlich, diejenige Zeit $\mathbf{t(i,k)}$ zu ermitteln, die das Modell i auf Knoten k für das Auslesen und Zurückschreiben seiner geschnittenen Facility-Werte benötigt hat:

$$t(i,k) = \frac{\sum_{j \in M_A(l),\ l \in K_A,\ l \neq k,\ i \neq j} n(i,j)}{n(k)} * t(k) \qquad (i \in M_A(k),\ k \in K_A)$$

[3]Bei einer testweisen oder virtuellen Modellverschiebung werden lediglich temporär diejenigen Datenstrukturen abgeändert, welche die Zuordnung der aktiven Teilmodelle zu den Knoten beinhalten. Im Gegensatz zu den logischen Modellverschiebungen werden bei einer testweisen Modellverschiebung keine Modellzustände übertragen.

Sei $\mathbf{t_{EVAL}(i,k)}$ des weiteren die gemessene Evaluationszeit des Modells i auf dem Knoten k. Dann läßt sich die Zykluszeit $\mathbf{t_Z(i,k)}$ des Modells i auf Knoten k wie folgt berechnen (die Zeit für den Datenaustausch über das Netzwerk geht hier nicht mit ein):

$$t_Z(i,k) = t_{EVAL}(i,k) + t(i,k) \qquad (i \in M_A(k),\ k \in K_A)$$

Nun kann die gewichtete Zykluszeit des Modells i, $\mathbf{t_{\overline{Z}}(i)}$, ermittelt werden. $t_{\overline{Z}}(i)$ ist knotenunabhängig und wird unter Einbeziehung des Lastindexes des Knotens k berechnet, auf dem i aktiv ist:

$$t_{\overline{Z}}(i) = \frac{t_Z(i,k)}{V_k} \qquad (i \in M_A(k),\ k \in K_A)$$

Wollte man nun im Rahmen des rekursiven Lastbalancierungsalgorithmus das Modell i testweise von Knoten k auf den Knoten k' verschieben (i muß dazu natürlich auf k' geladen sein), so ließe sich die Zykluszeit von i auf k' wie folgt vorhersagen:

$$t_Z(i,k') = t_{\overline{Z}}(i) * V_{k'} \qquad (i \in M,\ k' \in K)$$

Sei $\mathbf{M'_A(k)}$ die Menge der aktiven Teilmodelle auf k nach einer testweisen Modellverschiebung. Dann läßt sich die neue Zykluszeit $\mathbf{t_C(k)}$ für den Knoten k auf folgende Weise vorhersagen:

$$t_C(k) = \sum_{i \in M'_A(k)} t_{\overline{Z}}(i) * V_k \qquad (k \in K)$$

Abschließend ergibt sich die parallele Zykluszeit $\mathbf{t_{\overline{C}}}$ aus der größten Knoten-Zykluszeit:

$$t_{\overline{C}} = \max_{k \in K}\ t_C(k)$$

4.1.2.3 Rekursiver Lastbalancierungsalgorithmus

Die Suche nach möglichen Modellverschiebungen erfolgt über einen Backtracking-Algorithmus. Dieser von J. Löser entwickelte Algorithmus ([Löser 1998], S. 30) bildet den Kern der Lastbalancierung und ist nachfolgend in Pseudonotation dargestellt (Algorithmus 5):

Algorithmus 5 Rekursiver Lastbalancierungsalgorithmus

```
time ⇐ predicted parallel cycletime
recursive function rec_dlb (depth,maxdepth)
n ⇐ worst node
for all m ∈ active models on node n do
  for all k ∈ Nodes : k ≠ n, m loaded on k do
    virtually move model m to node k
    t ⇐ new predicted parallel cycletime
    if t * (1 + depth * MOVE_OFFSET) < time then
      time ⇐ t
      save moves done up to now
    end if
    if depth < maxdepth then
      rec_dlb (depth + 1, maxdepth)
    end if
    virtually move model m back to node n
  end for
end for
```

Der Algorithmus versucht, ausgehend von der aktuellen Verteilung der aktiven Teilmodelle, die parallele Zykluszeit $t_{\overline{c}}$ zu verbessern. Hierzu werden rekursiv die Knoten n mit der größten Zykluszeit $t_C(n)$ betrachtet (*worst node*). Nun wird für alle aktiven Modelle m aus $M_A(n)$ untersucht, ob sich die parallele Zykluszeit $t_{\overline{C}}$ durch testweises Verschieben von m auf einen anderen Knoten verbessert. Damit die aus der virtuellen Modellverschiebung resultierende neue Modellverteilung als besser akzeptiert wird, muß die parallele Zykluszeit der neuen Verteilung mindestens um einen bestimmten Betrag kleiner sein als das bisherige $t_{\overline{C}}$. Dieser Betrag hängt von der aktuellen Rekursionstiefe (*depth*) und einem Parameter *MOVE_OFFSET* ab. *MOVE_OFFSET* gibt an, um wieviel Prozent sich die parallele Zykluszeit mindestens verbessern muß, damit die testweise Modellverschiebung akzeptiert werden kann. Der Defaultwert liegt bei fünf Prozent.
Der Parameter *maxdepth* gibt die maximale Rekursionstiefe des Algorithmus an. Er bestimmt damit auch gleichzeitig die maximale Anzahl der möglichen Modellverschiebungen, da in jedem Rekursionsschritt jeweils nur ein Modell verschoben werden kann.
Der Algorithmus gibt am Ende eine Liste der durchzuführenden Modellverschiebungen zurück. Es ist aber auch durchaus möglich, daß keine Modellverteilung gefunden wird, die besser als die Ausgangsverteilung ist. In diesem Fall wäre die Liste leer.

4.2 Einflußmöglichkeiten der Lastbalancierung

Im experimentellen Vorfeld sollte untersucht werden, welche Parameter den Ablauf der dynamischen Lastbalancierung beeinflussen können. Abbildung 4.2 zeigt eine Übersicht über diese Einflußmöglichkeiten:

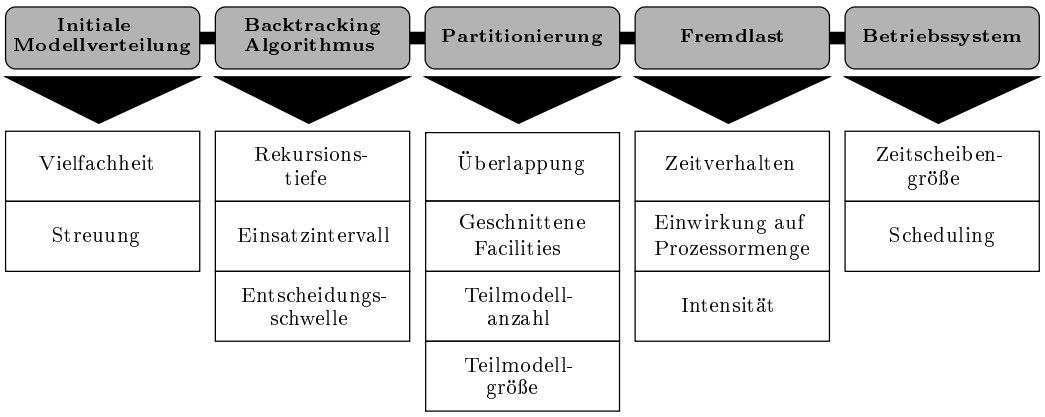


Abbildung 4.2: Einflußmöglichkeiten der Lastbalancierung

4.2.1 Partitionierung

Eine Partition, bestehend aus einer bestimmten Anzahl von Teilmodellen, läßt sich anhand folgender Eigenschaften charakterisieren:

- **Überlappung:** Die Überlappung von Teilmodellen einer Partition ist auf die Überlagerung der zugrundeliegenden Cones zurückzuführen. Die Logikbereiche im Überlappungsgebiet werden in der parallelen Simulation zwangsläufig mehrfach ausgewertet. Um diese unnötigen Mehrfachevaluierungen zu minimieren, sollten die Überlappungsgebiete möglichst klein sein.

- **Geschnittene Facilities:** Die Werte der im Rahmen der Partitionierung entstehenden geschnittenen Facilities müssen nach jedem simulierten Taktzyklus zwischen allen Slave-Knoten ausgetauscht werden. Eine Senkung der Anzahl der geschnittenen Facilities kann daher zu einer deutlichen Erhöhung der Simulationsgeschwindigkeit führen.

- **Teilmodellanzahl:** Eine hohe Anzahl von Teilmodellen vergrößert die Möglichkeiten des Lastbalancierungsalgorithmus bei der Suche nach günstigeren Modellverteilungen und erlaubt damit auch eine bessere Reaktion auf Fremdlasteinflüsse. Andererseits zieht eine große Teilmodellanzahl in der Regel auch eine hohe Zahl geschnittener Facilities nach sich.

- **Teilmodellgröße:** Die Größe eines Teilmodells wird durch die Anzahl der enthaltenen Logikelemente bestimmt. Bei kleinen Teilmodellen ist darauf zu achten, daß deren Zykluszeit möglichst groß im Vergleich zur Zeitscheibengröße des Betriebssystems ist[4]. Ein weiterer Aspekt ist die Frage nach der Erzeugung möglichst gleich großer Teilmodelle: Für den dlbSIM[5] ist die Homogenität der Teilmodelle von eher untergeordneter Bedeutung, da der Lastbalancierungsalgorithmus durch Modellverschiebungen in der Lage ist, auch bei unterschiedlichen Teilmodellgrößen auf allen Slave-Knoten ein ausgewogenes Verhältnis der simulationsspezifischen Last zu erzeugen.

4.2.2 Initiale Modellverteilung

Die initale Modellverteilung legt fest, welche Teilmodelle zu Beginn der Simulation auf welchen Slave-Knoten geladen werden. Die Teilmodelle sollen dabei so verteilt werden, daß der Lastbalancierungsalgorithmus diese später möglichst flexibel verschieben kann. Somit bestimmt die initale Modellverteilung von vornherein das Potential der möglichen Teilmodellverschiebungen. Der zugrundeliegende Verteilungsalgorithmus hat die Aufgabe, zu einer gegebenen Knoten- und Teilmodellanzahl eine Zuordnung dieser Teilmodelle zu den Knoten zu finden, wobei alle Knoten gleich viele Teilmodelle erhalten sollen.

[4] Zur Ermittlung der Evaluierungszeiten der Teilmodelle in einem Simulationsintervall werden sukzessive die zur Simulation eines Taktzyklusses benötigten Zeiten gemessen und über die Länge des Simulationsintervalls aufsummiert. Liegt hierbei die Dauer eines Taktzyklusses im Bereich einer Zeitscheibe, so können die gemessenen Zykluszeiten fehlerbehaftet sein (siehe auch [Löser 1998], S.33).

[5] Auch bei Verwendung des parallelMVLSIM, welcher nur ein (aktives) Teilmodell pro Slave-Knoten besitzt, ist die Erzeugung gleich großer Teilmodelle nicht immer sinnvoll, da bei der Forderung nach ausgewogener Logik meist auch die Anzahl der geschnittenen Facilities zunimmt.

Die „Qualität“ einer initialen Modellverteilung läßt sich anhand folgender Eigenschaften bewerten:

- **Vielfachheit:** Die Vielfachheit gibt an, wie oft jedes Teilmodell geladen werden soll. Die Modelle werden hierbei natürlich auf verschiedenen Slave-Knoten geladen. Eine große Vielfachheit ermöglicht mehr Freiheiten bei der Lastverschiebung, erhöht aber auch den Speicherverbrauch. Die maximale Vielfachheit wird durch die Anzahl der Slave-Knoten begrenzt.

- **Streuung:** Die Streuung ist ein allgemeinerer, umfassender Begriff zur Bewertung einer Teilmodellverteilung. Zum einen ist ist eine Verteilung als „günstig“ einzuschätzen, wenn es zu einem Knoten möglichst viele andere Knoten mit gemeinsamen Teilmodellen gibt. Hierbei darf es jedoch nicht zur *Clusterung* kommen, das heißt es sollte vermieden werden, daß sich Gruppen von Knoten bilden, die nur untereinander Teilmodelle austauschen können. Der zugrundeliegende Verteilungsalgorithmus unterbindet die Entstehung derartiger Cluster. Er sorgt weiterhin dafür, daß jedes Teilmodell mindestens zwei mal geladen wird (minimale Vielfachheit) und daß jeder Knoten mindestens zwei Teilmodelle besitzt.

Vor Simulationsbeginn müssen auf jedem Knoten die aktiven Teilmodelle festgelegt werden. Da angenommen wird, daß die Evaluierung eines jeden Teilmodells in etwa die gleiche Zeit benötigt und alle beteiligten Knoten relativ homogen sind, soll jeder Slave-Knoten zu Beginn die gleich Anzahl aktiver Teilmodelle zugewiesen bekommen. Diese Forderung läßt sich natürlich nur dann erfüllen, wenn die Teilmodellanzahl ein Vielfaches der Knotenzahl ist.

4.2.3 Fremdlast

Das wichtigste Kriterium zur Bewertung des dlbSIM ist dessen Verhalten unter Fremdlasteinwirkung. Die Art einer Lasteinwirkung läßt sich anhand folgender Punkte beschreiben:

- **Zeitverhalten:** Erfolgt die Lasteinwirkung nur für kurze Zeit (und dafür eventuell häufiger) oder über einen längeren Zeitraum hinweg ?

- **Einwirkung auf Prozessormenge:** Erfolgt die Fremdlasteinwirkung nur auf einem Knoten oder auf mehreren Knoten gleichzeitig ?

- **Intensität:** Erfolgt die Lasteinwirkung durch einen oder mehrere fremde Prozesse ?

4.2.4 Backtracking-Algorithmus

Die Arbeitsweise des rekursiven Lastbalancierungsalgorithmus (vgl. Algorithmus 5) läßt sich durch folgende Parameter steuern:

- **Rekursionstiefe:** Die Rekursionstiefe begrenzt die Anzahl der möglichen Teilmodellverschiebungen bei der Suche nach besseren Modellverteilungen. Wird die Rekursionstiefe groß genug gewählt, so erfolgt eine vollständige Analyse des Suchraumes. Aus Performancegründen ist dies jedoch nicht möglich, da die maximal erlaubte Laufzeit des Lastbalancierungsalgorithmus durch die Zeit begrenzt wird, welche die Slaves zur Simulation eines Simulationsintervalles benötigen. Bei der Wahl hoher Rekursionstiefen ist außerdem zu bedenken, daß jede Teilmodellverschiebung eine recht große Simulationsunterbrechung (von bis zu 2 Sekunden) darstellt.

- **Einsatzintervall:** Das Einsatzintervall für den Lastbalancierungsalgorithmus entspricht der Dauer eines Simulationsintervalls. Bei Verwendung des Defaultwertes von 1000 Zyklen für ein Simulationsintervall wird der Backtracking-Algorithmus etwa einmal pro Minute aktiviert. Für sehr große Teilmodelle muß die Zyklenzahl pro Simulationsintervall (<*dlbinterval*>) unter Umständen niedriger gewählt werden, um auf Fremdlasteinflüsse noch hinreichend schnell reagieren zu können. Bei sehr kurzen Einsatzintervallen ist zu beachten, daß dies auch eine Vergrößerung des Overheads für die Lastdatenübertragung der Slaves an den Master nach sich zieht.

- **Entscheidungsschwelle:** Die Entscheidungsschwelle (*MOVE_OFFSET*) gibt an, um wieviel Prozent die derzeit beste parallele Zykluszeit bei einer testweisen Modellverschiebung mindestens unterboten werden muß, damit diese Verschiebung akzeptiert wird. Sie läßt sich zur Dämpfung der Dynamik des Backtracking-Algorithmus verwenden: Eine hohe Entscheidungsschwelle wird eine Modellverschiebung nur dann zulassen, wenn diese eine signifikante Verbesserung der erwarteten parallelen Zykluszeit verspricht, währenddessen ein Wert von unter 5 Prozent unter Umständen zu vielen, „weniger guten“ Modellverschiebungen führen kann.

4.2.5 Betriebssystem

Zu guter Letzt nimmt auch das Betriebssytem über die zugrundeliegende **Scheduling-Strategie** und die **Zeitscheibengröße** Einfluß auf die parallele Simulation. Unter *AIX* beträgt die Länge einer Zeitscheibe 10 Millisekunden.

4.3 Experimente zum dlbSIM mit aktivierter Lastbalancierung

4.3.1 Die Testumgebung

Für die Experimente stand ein kleines Cluster aus drei heterogenen *RS/6000* Workstations zur Verfügung. Die einzelnen Knoten lassen sich hardwareseitig wie folgt charakterisieren:

- **Workstation 1 (W_1):** 132 MHz *Power2*-CPU, 2 GB RAM.
- **Workstation 2 (W_2):** 2 x *PowerPC 604* mit je 233 MHz, 1 GB RAM.
- **Workstation 3 (W_3):** 25 MHz *Power*-CPU, 64 MB RAM.

Das Geschwindigkeitsverhältnis der Knoten bezüglich eines sequentiellen MVLSIM-Laufes beträgt: $11(W_1) : 1(W_2) : 1{,}8(W_3)$. Der Grund für die überraschend niedrige Performance von W_2 ist darin zu vermuten, daß der MVLSIM-Code für die *Power*-Architektur optimiert wurde.
Die Knoten sind über das universitätsinterne 10 MBit Ethernet-Netzwerk miteinander verbunden. Daher konnte das Auftreten von Kommunikationsengpässen zwischen den Workstations nie vollständig ausgeschlossen werden. Durch eine Verlagerung der Simulationsläufe in die Nachtstunden war es jedoch möglich, störende Netzwerkeinflüsse weitesgehend zu umgehen. Auf den drei Workstations selbst bestand für die Zeit der Simulationsläufe exklusiver Zugriff, so daß keine anderen Nutzer unvorhersehbare Fremdlast erzeugen konnten.
Softwareseitig stand auf allen Knoten das Betriebssystem *AIX 4.3* sowie das *IBM Parallel Environment* zur Verfügung.
Sämtliche Simulationsläufe wurden mit 3 Slave-Prozessen durchgeführt, wobei jede Workstation einen Slave-Prozeß zugeordnet bekam (Slave S_i auf Workstation W_i, $i = 1, 2, 3$). Aufgrund der höheren Leistungsfähigkeit von W_1 wurde diesem Knoten zusätzlich der Master-Prozeß zugewiesen.

4.3.2 Partitionierung

Für die parallele Simulation wurde das Proto des *S/390* Prozessors *Monet* in acht Teilprotos partitioniert. Die Verwendung des großen Monet-Chips stellt sicher, daß die Zykluszeiten der resultierenden Teilmodelle trotz der vergleichsweise hohen Anzahl von Teilprotos deutlich über der Dauer einer Zeitscheibe liegen. Die Partitionierung wurde wiederum mit Hilfe der *PEnv* unter Verwendung des *STEP* und der evolutionären Algorithmen durchgeführt.

Tabelle 4.1 zeigt für die entstandenen Teilmodelle jeweils deren Größe, die Anzahl der zugrundeliegenden Logikelemente (Boxen) sowie die Anzahl der geschnittenen Facilities.

Teilmod. Nummer	**Teilmod. Größe**	**Boxanzahl Teilproto**	**Anzahl geschn. Input-Facilities**	**Anzahl geschn. Output-Facilities**
1	9,2 MB	400.000	6798	5486
2	7,5 MB	353.000	5475	6881
3	6,8 MB	343.000	2816	2938
4	6,1 MB	344.000	2021	2734
5	11,0 MB	471.000	5097	3029
6	8,9 MB	290.000	2334	5408
7	9,5 MB	330.000	6265	2305
8	9,2 MB	314.000	2594	1752

Tabelle 4.1: Partitionierung des *S/390*-Prozessors *Monet* in 8 Teilmodelle

Addiert man die Boxanzahlen sämtlicher Teilprotos, so ergeben sich etwas mehr als 2.8 Millioen Logikelemente. Das Ausgangsproto besitzt knapp 2.7 Millionen Logikelemente. Somit existieren gut 100.000 Boxen, welche aufgrund von Coneüberlappungen mehrfach evaluiert werden müssen.

4.3.3 Initiale Modellverteilung

In Tabelle 4.2 ist die initiale Verteilung der acht Teilmodelle ($M_1 \ldots M_8$) dargestellt. Jeder Slave-Knoten erhält hierbei sechs Teilmodelle. Die Vielfachheit, mit der die einzelnen Modelle verteilt wurden, liegt zwischen zwei und drei. Damit ist dem Lastbalancierungsalgorithmus ein ausreichendes Potential für Modellverschiebungen gegeben.

	$\mathbf{M_1}$	$\mathbf{M_2}$	$\mathbf{M_3}$	$\mathbf{M_4}$	$\mathbf{M_5}$	$\mathbf{M_6}$	$\mathbf{M_7}$	$\mathbf{M_8}$
Slave S_1	*	*	*	*	*	*		
Slave S_2	*	*	*		*		*	*
Slave S_3	*	*		*		*	*	*

Tabelle 4.2: Initiale Verteilung der Teilmodelle auf die Slave-Knoten

Diese Modellverteilung blieb in allen Experimenten unverändert. Im Vorfeld wurde auch der Versuch unternommen, alle Modelle mit Vielfachheit drei

zu verteilen. Dann besäße jeder Slave-Knoten sämtliche Teilmodelle und es würde keinerlei Einschränkungen bei Modellverschiebungen geben. Aufgrund der geringen Hauptspeicherausstattung von Workstation 3 (64 MB RAM) war dieser Rechner während eines Testlaufes allerdings fast ausschließlich mit dem Aus- und Einlagern von Speicherseiten auf Festplatte beschäftigt, so daß die Testergebnisse unbrauchbar wurden und obiger Modellverteilung der Vorzug zu geben war.
Tabelle 4.3 zeigt für jedes Teilmodell die Nummer des Slave-Knotens, auf welchem es zu Simulationsbeginn aktiv war. Mit Ausnahme des Experiments *„Load 4 without dlb“* besitzt die dargestelle initale Verteilung der aktiven Teilmodelle für alle Experimente Gültigkeit.

	$\mathbf{M_1}$	$\mathbf{M_2}$	$\mathbf{M_3}$	$\mathbf{M_4}$	$\mathbf{M_5}$	$\mathbf{M_6}$	$\mathbf{M_7}$	$\mathbf{M_8}$
Slave S_1	*			*				
Slave S_2			*		*		*	
Slave S_3		*				*		*

Tabelle 4.3: Verteilung der aktiven Teilmodelle zu Simulationsbeginn

4.3.4 Fremdlast

Der Einfluß „störender Prozesse“ wurde durch ein kleines Hilfsprogramm namens *loadnet* realisiert. Für jeden zu simulierenden Fremdlastprozeß generiert dieses Programm eine leere Endlosschleife. Das heißt, die Erzeugung einer zusätzlichen Last von 2 würde zur Erzeugung zweier neuer Prozesse führen, die lediglich in einer Endloschleife Rechenzeit verbrauchen. Auf Wunsch entfernt *loadnet* dann wieder die Fremdlastprozesse. *Loadnet* selbst benötigt dabei so gut wie keine Rechenzeit.

4.3.4.1 Intensität

Die nachfolgenden vier Experimente untersuchen das Lastbalancierungsvermögen des dlbSIM bei verschiedenen Fremdlastintensitäten. In allen Experimenten wurden 30000 Taktzyklen des *Monet*-Modells simuliert. Die Erzeugung zusätzlicher Fremdlast setzte stets nach 10000 Zyklen ein und wurde über einen Zeitraum von 10000 Zyklen aufrechterhalten. Die Länge eines Simulationsintervalls betrug 1000 Taktzyklen. Der rekursive Lastbalancierungsalgorithmus wurde mit einer Rekursionstiefe von 3 aufgerufen und die Entscheidungsschwelle (*MOVE_OFFSET*) lag bei 5%.

Experiment „No Load“
In diesem Experiment wurde noch keine zusätzliche Fremdlast generiert, das heißt ausgehend von der initialen Verteilung der aktiven Teilmodelle aus Tabelle 4.3 wurden bei aktivierter Lastbalancierung einfach nur 30000 Taktzyklen simuliert.
Abbildung 4.3 zeigt die benötigten Simulationszeiten für jedes Simulationsintervall. Die abgetragenen Zeiten sind die tatsächlich benötigen Laufzeiten (*REAL-Time*) zur vollständigen Abarbeitung eines Simulationsintervalls: Sie umfassen sowohl die Zeit zur Ausführung des parallelen Clock-Cycle-Algorithmus (Schritte *Clock*, *Get/Put* und *Transfer*) als auch die Zeit zur Durchführung von (logischen) Teilmodellverschiebungen. Eine Modellverschiebungsphase beinhaltet in der Regel mehrere Teilmodellverschiebungen und nahm, je nach Belastung des Netzwerkes, zwischen 1 bis 2 Sekunden in Anspruch.
Die Simulationszeit für das erste Simulationsintervall (die ersten 1000 Zyklen) beträgt 460 s. Nach 4 Modellverschiebungsphasen (nach 2000, 4000, 6000 und 8000 Zyklen) beträgt die Zeit für ein Simulationsintervall nur noch 210 s. Diese Zeit blieb bis zum Ende der Simulation konstant und es traten auch keine weiteren Modellverschiebungen auf. Anhang C enthält zu fast allen experimentellen Untersuchungen eine Tabelle, in der für alle Teilmodelle und alle Simulationsintervalle die Nummer desjenigen Slave-Knotens zu finden ist, auf welchem das entsprechende Teilmodell im angegebenen Simulationsintervall aktiv war. Nach der letzten Modellverschiebungsphase besitzt S_1 demzufolge fünf aktive Teilmodelle (1, 2, 3, 5, 6), S_2 ein aktives Teilmodell (7) und Slave S_3 zwei aktive Modelle (4 und 8). Diese durch den Lastbalancierungsalgorithmus gefundene finale Verteilung der aktiven Teilmodelle deckt sich mit dem zuvor erwähnten Geschwindigkeitsverhältnis der Knoten bezüglich eines sequentiellen MVLSIM-Laufes von $11(W_1) : 1(W_2) : 1{,}8(W_3)$.
Aus den Resultaten wird deutlich, daß der dlbSIM in der Lage ist, ungünstige initiale Modellverteilungen auf einem heterogenen System wieder auszugleichen.

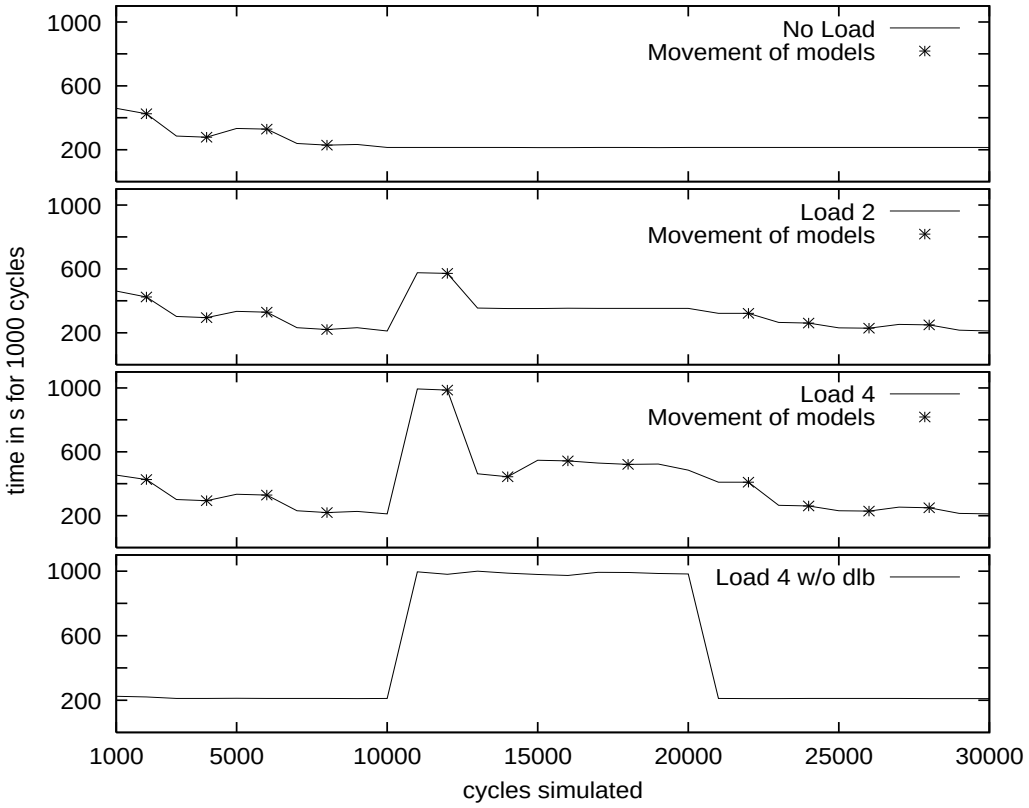


Abbildung 4.3: Simulationszeiten bei verschiedenen Fremdlastintensitäten

Experiment „Load 4 without dlb“
Dieser Testlauf wurde mit der finalen Teilmodellverteilung des vorherigen Experiments gestartet. Auf dem Knoten W_1 wurde bei deaktivierter Lastbalancierung nach 10000 Zyklen eine zusätzliche Last von 4 erzeugt, welche nach 20000 Zyklen wieder entfernt wurde. Der Einfluß der Fremdlast auf die Simulationszeit ist deutlich in Abbildung 4.3 erkennbar.

Experimente „Load 2“ und „Load 4“
In beiden Experimenten wurde bei aktiver Lastbalancierung auf Knoten W_1 eine zusätzliche Last von 2 (bzw. 4) erzeugt. Die resultierenden Simulationszeiten sind wiederum in Abbildung 4.3 dargestellt. Während der ersten 10000 Zyklen ergaben sich die gleichen Simulationszeiten wie im Experiment *„No Load“*. Auch ist die Verteilung der aktiven Teilmodelle nach 10000 Zyklen in beiden Fällen identisch mit der finalen *„No Load“*-Teilmodellverteilung.

Unter der Fremdlast von 2 (4) haben sich die Simulationszeiten von 210 s auf 580 s (990 s) erhöht. Nach 11000 Zyklen erhält der Master neue Lastdaten von den Slaves, welche die veränderte Lastsituation auf S_1 zum Ausdruck bringen. Im darauffolgenden Simulationsintervall wertet der Master (parallel zur Simulation) die neuen Lastdaten aus. Im Ergebnis der Auswertung wird nach 12000 Zyklen die erste Modellverschiebungsphase unter Fremdlast durchgeführt, was im Falle des Experiments *„Load 2“* zu einer sofortigen, dauerhaften Absenkung der Simulationszeit auf 350 s führt. Im Experiment *„Load 4“* sind noch drei weitere Modellverschiebungsphasen notwendig, ehe sich die Simulationszeit unter Lasteinwirkung bei 490 s stabilisiert.
Nach dem Entfernen der Fremdlast initiiert der Master erneut einige Modellverschiebungsphasen. Letztendlich stellt sich nach 30000 Zyklen wieder exakt das gleiche Bild wie unmittelbar vor der Lasterzeugung ein. Dies trifft sowohl auf die Simulationszeiten als auch auf die Verteilung der aktiven Teilmodelle zu.
Zur weiteren Veranschaulichung ist in Abbildung 4.4 (4.5) die detailierte Zusammensetzung der Simulationszeiten auf allen drei Workstations sowie die momentane Verteilung der aktiven Teilmodelle für ausgewählte Simulationsintervalle dargestellt. Alle gemessen Zeiten spiegeln die tatsächlich vergangene REAL-Time wider und wurden über die Länge des entsprechenden Simulationsintervalls sukzessive aufsummiert. Folgende Simulationsintervalle wurden zur Darstellung ausgewählt:

- **Initial state (nach 1000 Zyklen):** Ausgangszustand nach Simulation des ersten Simulationsintervalls.
- **Adaption (nach 10000 Zyklen):** Zustand nach Anpassung der Modellverteilung an die Heterogenität des Workstationclusters.
- **Extra load (nach 11000 Zyklen):** Unmittelbarer Zustand nach Hinzunahme der Fremdlast.
- **Adaption (nach 20000 Zyklen):** Zustand kurz vor dem Entfernen der Fremdlast. Die Modellverteilung ist an die Fremdlastsituation angepaßt.
- **Load removed (nach 21000 Zyklen):** Unmittelbarer Zustand nach dem Entfernen der Fremdlast.
- **Final state (nach 30000 Zyklen):** Finalzustand. Es stellt sich der gleiche Zustand wie unmittelbar vor Beginn der Lasteinwirkung ein.

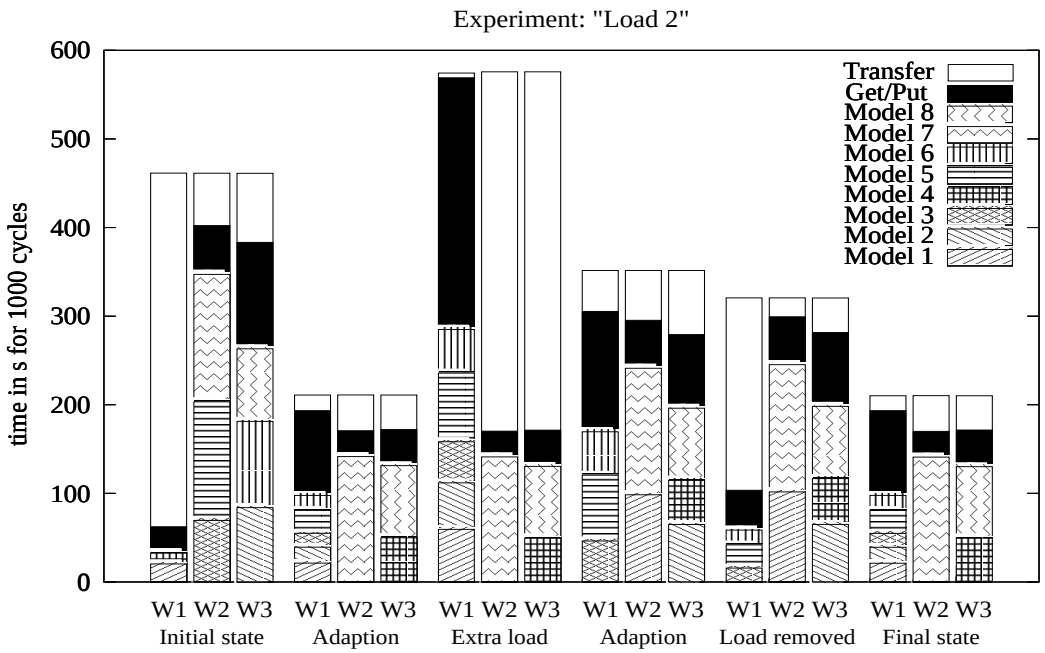


Abbildung 4.4: Detailierte Simulationszeiten und Teilmodellverteilungen im Experiment „Load 2“

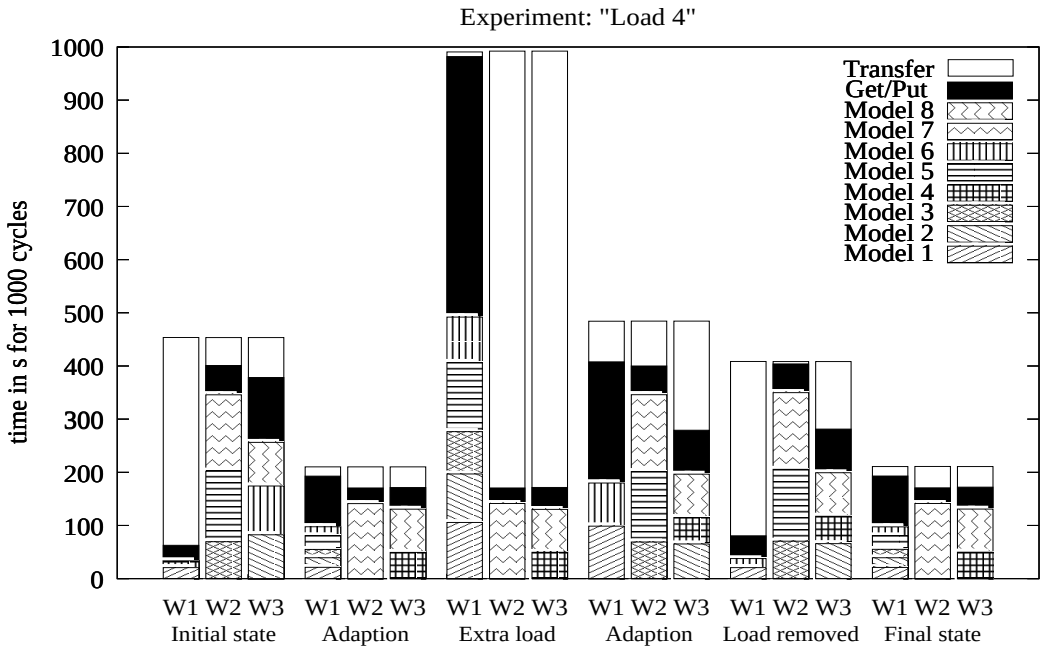


Abbildung 4.5: Detailierte Simulationszeiten und Teilmodellverteilungen im Experiment „Load 4“

Anhand der beiden Abbildungen 4.4 und 4.5 sind die Auswirkungen der Fremdlasteinwirkung sowie die Reaktion des Lastbalancierungsalgorithmus optisch nachvollziehbar: Im Experiment *„Load 2“* gibt W_1 unter Fremdlasteinfluß je ein Teilmodell an W_2 und W_3 ab. Bei einer Fremdlast von 4 gehen (nach mehreren Modellverschiebungsphasen) schließlich zwei Teilmodelle an W_2 und ein Teilmodell an W_3.

4.3.4.2 Einwirkung auf Prozessormenge

Aufgrund der eingeschränkten Leistungsfähigkeit des zur Verfügung stehenden Workstationclusters wurde von der gleichzeitigen Fremdlasterzeugung auf mehreren Knoten abgesehen. Allerdings wurden drei Experimente mit Fremdlasteinluß nur auf W_2 bzw. W_3 durchgeführt. Diese sollen hier beschrieben werden. Die Randbedingungen der vorangegangenen Untersuchungen behalten weiterhin Gültigkeit, mit der Ausnahme, daß die Simulation nun 40000 Taktzyklen umfaßt, da sich nach 30000 Zyklen mitunter noch kein stabiler Finalzustand eingestellt hatte.

Experiment „Load 1 on W_3“
In diesem Experiment wurde eine zusätzliche Last von 1 auf Knoten W_3 erzeugt. Die Abbildungen 4.6 und 4.7 [6] zeigen die Ergebnisse. Die ersten 10000 Zyklen verlaufen analog zu den vorangegangenen Experimenten. Nach Hinzunahme der Last steigt die Simulationszeit von 210 s auf 360 s an. Mit der Abgabe von Teilmodell 4 an W_1 nach 12000 Zyklen kann die Simulationszeit unter Fremdlast auf 230 s abgesenkt werden. Unmittelbar nach dem Entfernen der Fremdlast nach 20000 Zyklen fällt die Simulationszeit auf 220 s. Letztendlich wandert Teilmodell 4 mit einer weiteren Modellverschiebung zurück auf W_3, so daß die Simulationszeit wieder 210 s beträgt.

[6] **Final state** bezieht sich jetzt auf den Zustand nach 40000 Taktzyklen.

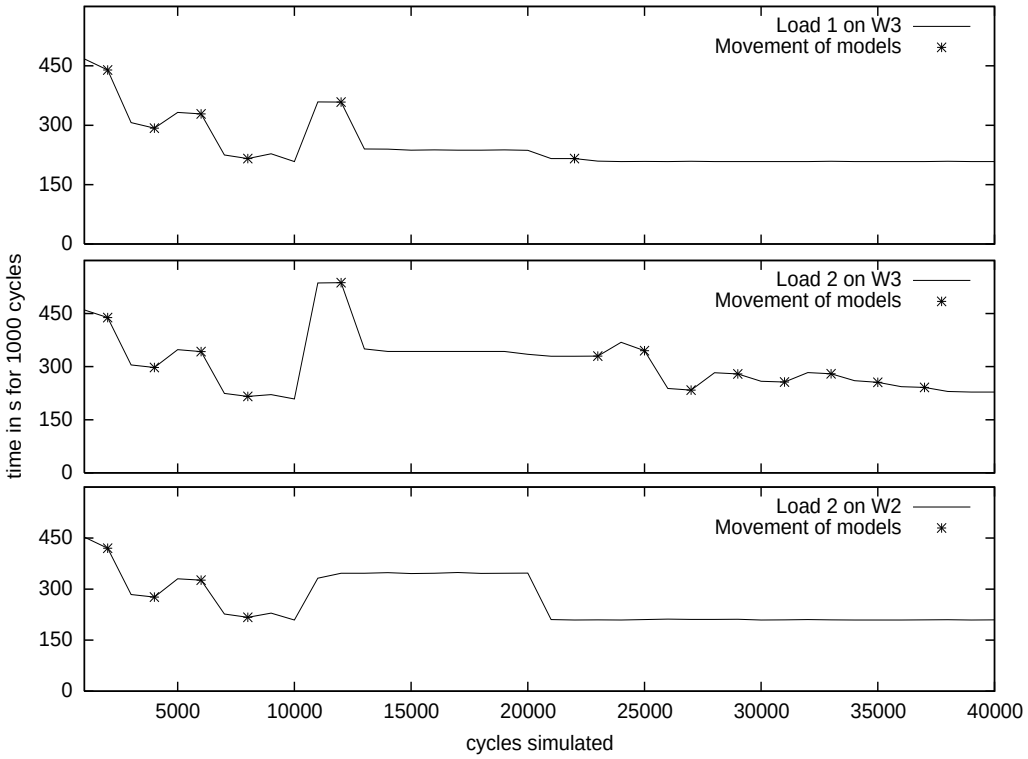


Abbildung 4.6: Simulationszeiten bei Fremdlasteinwirkung auf verschiedenen Slave-Knoten

Experiment „Load 2 on W_3“
Die Ergebnisse dieses Experiments sind in den Abbildungen 4.6 sowie 4.8 zu finden. Die zusätzliche Fremdlast von 2 bewirkt einen Anstieg der Simulationszeit von 210 s auf 550 s. Nach 12000 Zyklen gibt W_3 Modell 4 an W_1 und Modell 8 an W_2 ab. Damit hat W_3 all seine aktiven Teilmodelle abgegeben und der Master ist gezwungen, zur Berechnung des Lastindexes für W_3 auf die Load-Zahlen des Betriebssystems zurückzugreifen. Dies könnte auch der Grund dafür sein, daß der Lastbalancierungsalgorithmus nach dem Entfernen der Fremdlast offenbar etwas orientierungslos ist und eine Flut von 8 Modellverschiebungsphasen initiiert, so daß sich die Simulationszeiten erst nach einiger Zeit wieder einpendeln (bei 230 s). Zu bemerken ist hier allerdings, daß sich die finale Modellverteilung von der Verteilung der Modelle kurz vor der Fremdlasteinwirkung unterscheidet: Zwar erhält W_3 das Teilmodell 4 von W_1 zurück, der Knoten W_2 behält jedoch das Teilmodell 8 und verschiebt stattdessen Modell 7 auf W_3.

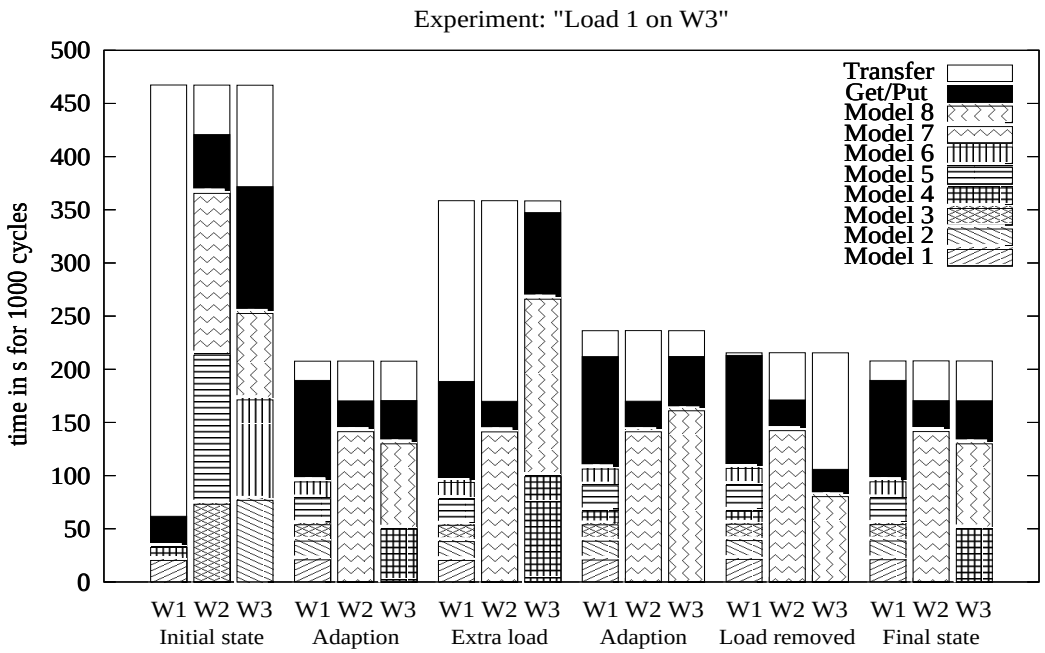


Abbildung 4.7: Detailierte Simulationszeiten und Teilmodellverteilungen im Experiment „Load 1 on W_3“

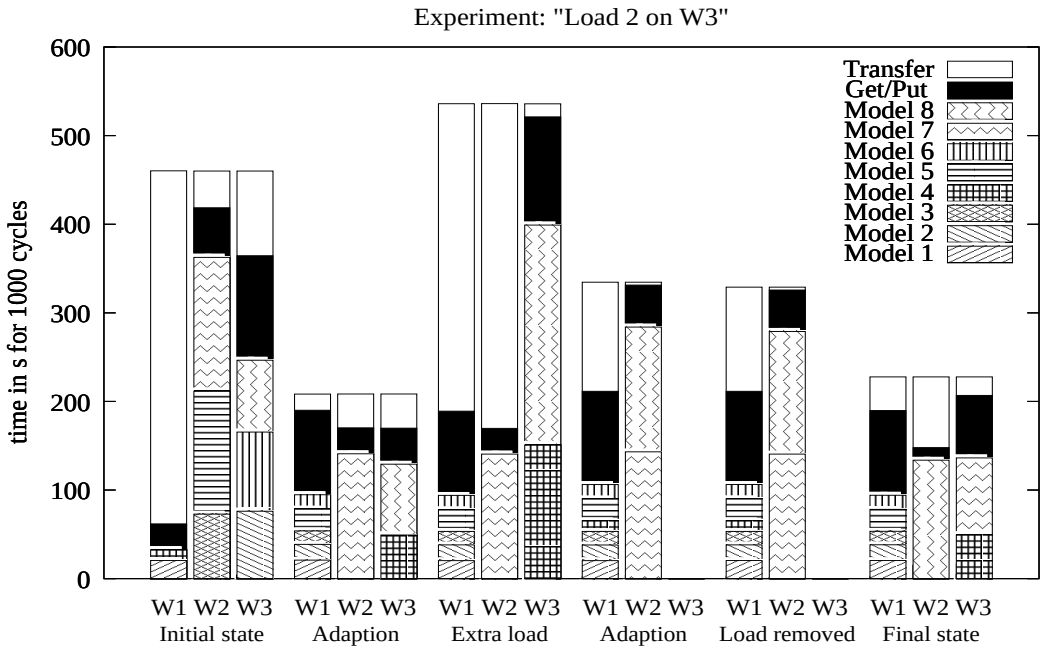


Abbildung 4.8: Detailierte Simulationszeiten und Teilmodellverteilungen im Experiment „Load 2 on W_3“

Experiment „Load 2 on W_2“
Das Ergebnis dieses Testlaufes ist in Abbildung 4.6 zu finden. Die ersten 10000 Zyklen zeigen das gewohnte Bild der Absenkung der Simulationszeit von 460 s auf 210 s. Es ist anzumerken, daß W_2 als „langsamster Knoten“ nach 10000 Zyklen lediglich ein aktives Teilmodell besitzt. Dieses gibt W_2 auch unter Lasteinfluß nicht her, so daß es nur zu einem Anstieg der Simulationszeit auf 350 s ohne Modellverschiebungen kommt. Nach Entfernen der Fremdlast fällt die Zeit wieder auf 210 s zurück.

4.3.4.3 Zeitverhalten

Die folgenden zwei Experimente untersuchen den Einfluß der Dauer der Fremdlasteinwirkung auf die Lastbalancierung. Es sollte herausgefunden werden, über welchen Zeitraum hinweg die Fremdlast aufrecht erhalten werden kann, ehe es zu Modellverschiebungen kommt. Hierzu wurde unter Verwendung von *loadnet* eine Last von 2 auf W_1 über mehrere Zeiträume von anwachsender Dauer (5 s bis 160 s) erzeugt. Die Länge eines Simulationsintervalls betrug wiederum 1000 Taktzyklen und auch die Paramter für den rekursiven Lastbalancierungsalgorithmus blieben unverändert (Rekursionstiefe 3, *MOVE_OFFSET* 5%).

Experiment „Shorter load intervals“
Dieses Experiment erstreckte sich über einen Zeitraum von insgesamt 60000 Zyklen. Nach Ablauf der ersten 10000 Taktzyklen stellt sich die gewohnte Zeit von 210 s pro Simulationsintervall ein. Abbildung 4.9 zeigt die Ergebnisse. (Die Sekundenangaben in den Klammern bezeichnen jeweils die Dauer der Lasteinwirkung.) Nun wird zu Beginn des 11. Simulationsintervalls (Zyklen 10001 ... 11000) erstmals für 5 s Fremdlast erzeugt, wodurch sich die Simulationszeit in diesem Intervall kaum merklich auf 215 s erhöht. Nach 20000 (30000, 40000, 50000) Zyklen erfolgt die Lasteinwirkung für 10 (20, 40, 60) Sekunden und die Simulationszeit steigt auf 215 (220, 235, 250) Sekunden an, ohne daß es jedoch zu Modellverschiebungen kommt.

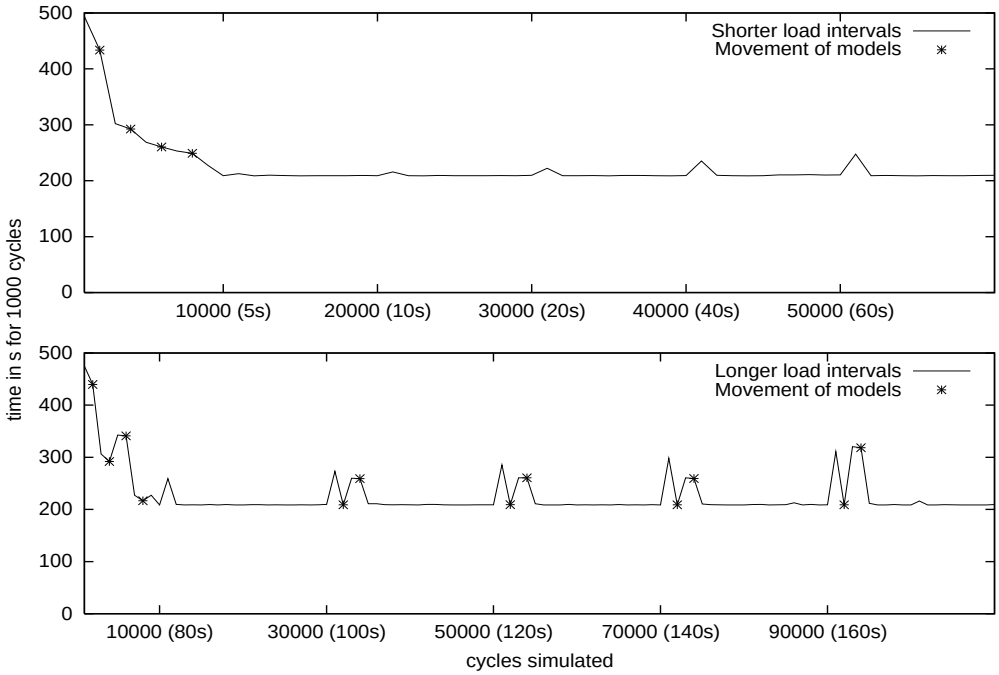


Abbildung 4.9: Simulationszeiten bei variierender Dauer der Fremdlasteinwirkung

Experiment „Longer load intervals“
Nachdem 60 *s* Lasteinwirkung noch keine Modellverschiebungen hervorrufen konnten, wurde ein analoges Experiment mit längeren Lastphasen über insgesamt 120000 Taktzyklen durchgeführt. Die Ergebnisse sind ebenfalls aus Abbildung 4.9 ersichtlich. Es wurden erneut 5 Fremdlastphasen von 80 (100, 120, 140, 160) Sekunden Länge nach 10000 (30000, 50000, 70000, 90000) Zyklen eingeleitet, die einen Anstieg der Simulationszeiten von 210 *s* auf 260 (275, 285, 300, 310) Sekunden zur Folge hatten. Da die Dauer der einzelnen Fremdlastphasen jedoch unterhalb der Dauer eines Simulationsintervalls lag, fielen die Simulationszeiten im Intervall nach der Lasteinwirkung stets wieder auf 210 *s* zurück.
Die durch 100 *s* Fremdlasteinfluß hervorgerufene Simulationszeiterhöhung auf 275 *s* veranlaßt den Lastbalancierungsalgorithmus erstmals zur Durchführung einer Modellverschiebung: Nach 31000 Zyklen erhält der Master ein Bild von der erhöhten Fremdlastsituation (welche zu diesem Zeitpunkt aber bereits nicht mehr vorliegt), so daß er sich nach 32000 Zyklen zu einer unnötigen Modellverschiebung entschließt, die bei Fortbestehen der Lasteinwirkung jedoch durchaus sinnvoll gewesen wäre.

Da bei einer Modellverschiebung die gesammelten Lastdaten invalidiert werden, bemerkt der Master hier immer noch nicht das Verschwinden der Fremdlast. Erst nach 33000 Zyklen erhält er die aktuellen Lastdaten und macht nach 34000 Zyklen die getätigte Modellverschiebung schließlich wieder rückgängig. In den nachfolgenden Phasen verlängerter Fremdlasteinwirkung wiederholt sich dieses Phänomen in analoger Weise.

4.3.5 Backtracking-Algorithmus

Im folgenden soll nun der Einfluß der verschiedenen Parameter des Backtracking-Algorithmus auf den Verlauf der Lastbalancierung untersucht werden. Die beschriebenen Experimente hatten einen Umfang von 30000 Taktzyklen. Die Lasteinwirkung erfolgte erneut nach 10000 Zyklen auf W_1. Nach 20000 Zyklen wurde die Fremdlast wieder entfernt.

4.3.5.1 Rekursionstiefe

Es wurden zwei Experimente im Stil von *„Load 2“*, jedoch mit maximalen Rekursionstiefen von 5 bzw. 7 durchgeführt. Die Ergebnisse waren in beiden Fällen nahezu identisch mit denen des *„Load 2“* -Testlaufes und sind daher nicht extra aufgeführt. Offenbar ist eine Erhöhung der Rekursionstiefe über den Defaultwert hinaus wenig sinnvoll, was sich mit den experimentellen Beobachtungen aus [Löser 1998] S.31 deckt, wonach bereits eine maximale Rekursionstiefe von 3 als ausreichend für die Erkennung und den Ausgleich von Lastungleichgewichten begutachtet wurde.

4.3.5.2 Einsatzintervall

Experiment „Simulation interval 500“
Auch dieses Experiment stellt eine leichte Modifikation des *„Load 2“*-Versuches dar: Die Länge eines Simulationsintervalls wurde hier auf 500 Taktzyklen verringert. Die Ergebnisse aus Abbildung 4.10 verdeutlichen gegenüber dem Experiment *„Load 2“* eine schnellere Reaktion des Lastbalancierungsalgorithmus auf Fremdlasteinflüsse (vgl. Abbildung 4.3). Zudem findet durch das kürzere Einsatzintervall zu Simulationsbeginn auch eine raschere Anpassung der Modellverteilung an die Heterogenität des Workstationclusters statt. Bezüglich der Verteilung der aktiven Teilmodelle gab es sowohl kurz vor Lasteinwirkung als auch am Simulationsende keine Unterschiede zum Experiment *„Load 2“*.

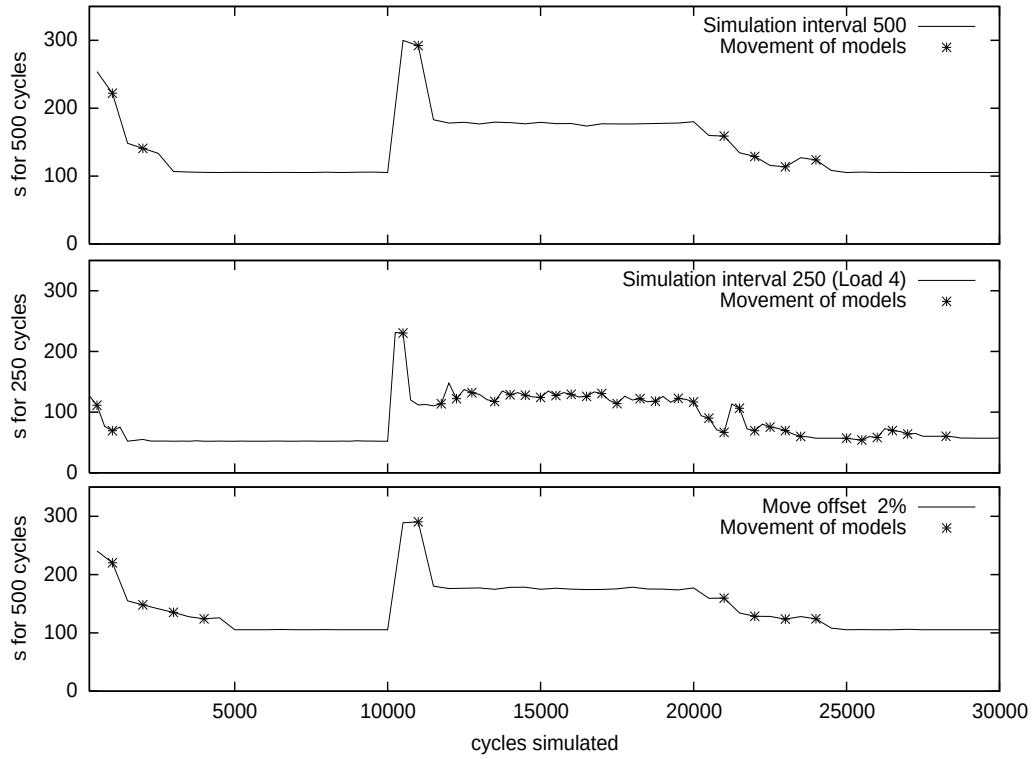


Abbildung 4.10: Simulationszeiten bei veränderten Parametern des rekursiven Lastbalancierungsalgorithmus

Experiment „Simulation interval 250“ (Load 4)
Im Vergleich zum vorangegangenen Experiment wurde hier die Länge des Einsatzintervalls weiter auf 250 Taktzyklen verkürzt. Aus Abbildung 4.10 wird eine noch schnellere Reaktion des Lastbalancierungsalgorithmus auf den Fremdlasteinfluß deutlich. Allerdings zieht das kurze Simulationsintervall bei Lasteinwirkung auch eine Flut von Modellverschiebungen nach sich, die bis zum Simulationsende nicht mehr abreißt.

4.3.5.3 Entscheidungsschwelle

Experiment „Move offset 2%“
Für diesen Versuch wurde das Experiment *„Simulation interval 500“* dahingehend modifiziert, daß die Größe der Entscheidungsschwelle für die Akzeptanz von Modellverschiebungen auf 2% herabgesetzt wurde.
Die Ergebnisse aus Abbildung 4.10 lassen im Vergleich zu *„Simulation interval 500“* eine leicht gestiegene Bereitschaft des Lastbalancierungsalgorithmus zur Durchführung von Modellverschiebungen vermuten, da innerhalb

der ersten 10000 Zyklen zwei zusätzliche Modellverschiebungsphasen stattgefunden haben. In vorangegangenen Experimenten mit einem Move Offset von 3% war diese Tendenz noch nicht erkennbar.
Inwieweit diese beiden zusätzlichen Modellverschiebungsphasen dem Zufall zuzuschreiben sind, muß durch weitere Experimente geklärt werden. Es ist allerdings schon jetzt feststellbar, daß für die vorliegende Testumgebung eine Verringerung der Entscheidungsschwelle keinen signifikanten Einfluß auf das Finden besserer Modellverteilungen ausübt.

Kapitel 5

Zusammenfassung der Ergebnisse

In diesem Kapitel werden abschließend alle wichtigen Ergebnisse dieser Arbeit zusammengefaßt. Weiterhin werden offene Aufgabenstellungen diskutiert und mögliche Richtungen für fortführende Arbeiten vorgeschlagen.

5.1 Integration des dlbSIM in eine Simulationsumgebung

Im Rahmen dieser Arbeit wurde der auf MVLSIM basierende parallele Logiksimulator dlbSIM zu Testzwecken in eine industrielle Simulationsumgebung integriert. Da die firmeninterne Weiterentwicklung des sequentiellen MVLSIM seit Fertigstellung des dlbSIM bereits in großem Umfang fortgeschritten war, mußte dlbSIM zunächst auf die aktuell gültige MVLSIM-Version umgestellt werden. Zur modularen Integration der aktuellen MVLSIM-Quellen war es erforderlich, die enge Verflechtung der parallelen Slavekomponente des dlbSIM mit dem zugrundeliegenden MVLSIM aufzulösen. Dies wurde insbesondere durch die Schaffung eines neuen Parallel-Simulator-APIs realisiert.
Leider war nach abgeschlossener Integration eine Deaktivierung der dynamischen Lastbalancierung notwendig, da MVLSIM sei einiger Zeit nur noch die Verwaltung eines Teilmodells erlaubt. Sollte diese Einschränkung in zukünftigen Versionen wieder aufgehoben werden, so wäre seitens des dlbSIM eine Reaktivierung der Lastbalancierung problemlos möglich.
Bei der Wahl der Partitionierungsumgebung fiel die Entscheidung zugunsten der *PEnv* aus, da diese sich gut in die vorhandene kommandozeilenorientierte Skriptumgebung eingliedert.

Für einen praktischen Einsatz wurde die *PEnv* auf das neue Listenfile-Format sowie die aktuelle PIF-Implementierung umgestellt.
Anhand von Experimenten wurde die korrekte Arbeitsweise des neuen dlbSIM bei deaktivierter Lastbalancierung überprüft und untersucht, inwieweit sich eine Senkung der Simulationszeit gegenüber MVLSIM bei der Simulation praxisrelevanter Modelle einstellt. Hierbei hat sich zunächst herausgestellt, daß ein sinnvoller Einsatz des dlbSIM erst bei verhältnismäßig großen Modellen mit deutlich mehr als 50000 Logikbausteinen möglich ist. Im Rahmen der parallelen Simulation des *S/390*-Prozessors *Monet*, welcher rund 2.7 Millionen Logikbausteine enthält, ließen sich unter Verwendung von 5 Slave-Knoten bereits Speedup-Werte von 2.5 erzielen.
In den Untersuchungen wurde deutlich, daß die zeitlichen Kosten des Zugriffs auf die geschnittenen Facilities im Allgemeinen recht hoch sind und in Ausnahmefällen bis zu 50% der Simulationszeit eines Slave-Knotens in Anspruch nehmen. Als Lösung könnte man bereits während des Partitionierungsprozesses, beispielsweise durch die Vergabe von Strafpunkten, versuchen, das Entstehen geschnittener Facilities zu erschweren. Alternativ wäre auch eine effizientere Implemtierung der Funktionen *ps_getfacdata()* und *ps_putfacdata()* in Betracht zu ziehen.
Mitunter traten in Folge einer unausgewogenen Verteilung der Simulationslast längere Wartezeiten im Vorfeld des Austausches der Werte der geschnittenen Facilities auf, da nicht alle Slave-Knoten gleichzeitig zur Kommunikation bereit waren. Bei deaktivierter Lastbalancierung besteht keine Möglichkeit des Ausgleichs derartiger Lastungleichgewichte. Daher wäre es eventuell sinnvoll, in künftigen Arbeiten an den Partitionierungsalgorithmen noch verstärkter darauf zu achten, für alle Modellblöcke die Summe von vorhergesagter Evaluationszeit und Austauschzeit der geschnittenen Facility-Werte möglichst gleich zu halten.

5.2 Untersuchungen zur Lastbalancierung des dlbSIM

Im zweiten Teil der Arbeit wurden experimentelle Untersuchungen zur dynamischen Lastbalancierung des dlbSIM durchgeführt. Um das Feature der Lastbalancierung nutzen zu können, mußte die zugrundeliegende aktuelle MVLSIM-Version des neuen dlbSIM gegen diejenge ausgetauscht werden, welche auch im originalen dlbSIM Verwendung fand.
Die Experimente wurden auf einem kleinen, lose gekoppelten Workstationcluster, bestehend aus 3 heterogenen Maschinen, durchgeführt. Im Verlaufe

der Untersuchungen hat sich bestätigt, daß das Lastbalancierungsfeature des dlbSIM die Simulationszeit unter Fremdlasteinfluß signifikant verkürzen kann. Des weiteren ist dlbSIM in der Lage, ungünstige initiale Modellverteilungen auf heterogenen Systemen durch Teilmodellverschiebungen zu kompensieren. Damit bieten sich neben speziellen Parallelrechnern auch preiswerte Workstationcluster für langlaufende Simulationsprozesse mit großen Schaltungsmodellen an.
Für einen praxisrelevanten Einsatz des dlbSIM auf heterogenen Workstationclustern wäre es zudem denkbar, die initiale Verteilung der aktiven Teilmodelle nicht statisch, sondern in Abhängigkeit von der Leistungsfähigkeit der einzelnen Workstations vorzunehmen. Durch eine Vorhersage der Teilmodellevaluierungszeiten während des Partitionierungsprozesses und deren Einbeziehung im Rahmen der initialen Auswahl aktiver Teilmodelle könnte so für jeden Slave-Knoten bereits zu Simulationsbeginn eine günstige individuelle Teilmodellbelegung ermittelt werden.
Die durchgeführten experimentellen Untersuchungen decken trotz der teilweise begrenzten Hardwareausstattung der zur Verfügung stehenden Workstations nahezu alle Einflußfaktoren der Lastbalancierung ab. Dennoch sind sie nur als Basis für die zielgerichtete Durchführung weiterer Untersuchungen zu verstehen. Diese sollten jedoch zur Minimierung unvorhersagbarer Randbedingungen vorzugsweise auf einem exklusiv nutzbaren Parallelrechner mit homogenen Knoten durchgeführt werden. Im Rahmen künftiger Untersuchungen wäre bei ausreichender Hauptspeicherkapazität beispielsweise eine Verteilung sämtlicher Teilmodelle einer Partition auf alle Slave-Knoten denkbar, so daß sich keinerlei Einschränkungen bei der Auswahl möglicher Modellverschiebungen ergeben würden. Im Falle der Verfügbarkeit entsprechend leistungsfähiger Maschinen wären des weiteren auch Experimente mit gleichzeitiger Fremdlasterzeugung auf mehreren Knoten bei vertretbarem zeitlichen Aufwand möglich.

Anhang A

Struktur der neuen Listenfiles

A.1 Binärversionen der Listenfiles

A.1.1 Binärversion der Crossreferenzliste (.pmod)

Bereich	Anzahl	Datentyp	Inhalt	Bedeutung
KOPF	1	1 x unsigned char	0x26	Dateikennung
		1 x unsigned char	0x05	Version
		8 x unsigned char	'17:53:34'	Erstellungszeit des pmod-Files
		8 x unsigned char	'11/10/99'	Erstellungsdatum des pmod-Files
		1 x unsigned int	G	Anzahl der globalen Facilities
		1 x unsigned int	P	Anzahl der parallelen Facilities
		1 x unsigned int	LN	Länge des Bereichs NAMEN
		1 x unsigned int	LS	Länge des Bereichs SYMBOL
		1 x unsigned int	LP	Länge des Bereichs PARFAC
NAMEN[1]	G	1 x unsigned int	L	Länge des nachfolgenden Names in Bytes
		L x unsigned char	'NAME' 0x00	nullterminierter Facility-Name
SYMBOL[1]	G	1 x unsigned int	A	Anzahl der zu dieser globalen Facility zugehörigen parallelen Facilities[2] .
		1 x unsigned int	RG	Anzahl der Zeilen (Rows) der globalen Facility
		1 x unsigned int	LG	Bitbreite (Length) der globalen Facility
		A x unsigned int	IP	Indizes der zugehörigen PARFAC-Strukturen

Tabelle A.1: Neues Format des pmod-Files für den Master (1. Teil)

Bereich	Anzahl	Datentyp	Inhalt	Bedeutung
PARFAC	P	1 x signed int	{-65536,..,65535}	Bitbreite/Bitposistion: {-2..-65536} -> Strand-Anzahl eines ungesplitteten Vektors (Vektoren der Länge 1 werden als Netze betrachtet) {-1} -> 'normales' Netz {0..65535} -> Bitposition eines Netzes in einem aufgesplitteten Vektor
		1 x unsigned char	M	Anzahl der Teilmodelle, in denen sich diese parallele Facility befindet. (Im Falle einer geschnittenen Facility oder durch Überlappung von Cones kann eine Facility mehreren Teilmodellen vorkommen.)
		1 x unsigned char	{0,1,2}	Facility-Kennung: {0} -> Array, {1} -> Netz, {2} -> Vektor
		1 x unsigned char	IS	Index der zugehörigen SYMBOL-Struktur
		M x unsigned char	NR	Nummern der Teilmodelle, auf denen sich diese parallele Facility befindet (beginnend mit 1)

Tabelle A.2: Neues Format des pmod-Files für den Master (2. Teil)

[1]Die Zuordung der NAMEN zu den SYMBOL-Einträgen der globalen Facilities ergibt sich aufgrund der gleichen Reihenfolge: Der i-te NAME gehört zum i-ten SYMBOL-Eintrag.

[2]In der Regel ist einer globalen Facility genau eine parallele Facility zugeordnet. Die Ausnahme sind die aufgesplitteten Vektoren, bei denen einzelne Strands während der Partitionierung unterschiedlichen Teilmodellen zugeordnet wurden. Für jeden Strand eines solchen aufgesplitteten Vektors existiert dann eine eigene parallele Facility.

A.1.2 Binärversion der Signalschnittliste ohne Simulatorindizes (.rmod)

Bereich	Anzahl	Datentyp	Inhalt	Bedeutung
KOPF	1	4 x unsigned char	'RR' 0x0000	Dateikennung
		1 x unsigned int	0x00000002	Version
		8 x unsigned char	'17:53:34'	Erstellungszeit des rmod-Files
		8 x unsigned char	'11/10/99'	Erstellungsdatum des rmod-Files
		1 x unsigned int	E	Anzahl der eingangsseitig geschnittenen Facilities dieses Teilmodells
		1 x unsigned int	A	Anzahl der ausgangsseitig geschnittenen Facilities dieses Teilmodells
		1 x unsigned int	T	Anzahl aller Teilmodelle
		1 x unsigned int	N	Anzahl der Einträge (Facility-Namen) im Bereich NAMEN
		1 x unsigned int	K	Länge des Bereichs KÖRPER in Bytes (als Offset für den Bereich NAMEN)
KÖRPER	E+A	1 x unsigned int	I	Index des zugehörigen Facility-Namens in der Liste der NAMEN
		1 x signed int	{-1,2..65536}	Bitbreite: -1 = Netz, 2..65536 = Vektor mit angegebener Anzahl von Strands (Vektor der Länge 1 = Netz)
		1 x unsigned char	M	Anzahl der Teilmodelle, die diese Facility speisen (für 'E') bzw. die von dieser Facility gespeist werden (für 'A')
		M x unsigned char	NR	Nummern dieser Teilmodelle (beginnend mit 1)
NAMEN	N	1 x unsigned int	L	Länge des nachfolgenden Names in Bytes
		L x unsigned char	'NAME' 0x00	nullterminierter Facility-Name

Tabelle A.3: Neues Format der rmod-Files für die Slaves

A.1.3 Binärversion der Signalschnittliste mit Simulatorindizes (.pmod)

Bereich	Anzahl	Datentyp	Inhalt	Bedeutung
KOPF	1	1 x unsigned char	0x1A	Dateikennung
		1 x unsigned char	0x02	Version
		8 x unsigned char	'17:53:34'	Erstellungszeit des Teilmodells
		8 x unsigned char	'11/10/99'	Erstellungsdatum des Teilmodells
		3 x unsigned int	0x...	Erstellungszeit (binär) des pmod-Files
		3 x unsigned int	0x...	Erstellungsdatum (binär) des pmod-Files
		1 x unsigned int	E	Anzahl der eingangsseitig geschnittenen Facilities dieses Teilmodells
		1 x unsigned int	A	Anzahl der ausgangsseitig geschnittenen Facilities dieses Teilmodells
		1 x unsigned int	T	Anzahl aller Teilmodelle
		1 x unsigned int	F	Anzahl der Facilities des originalen, sequentiellen Modells = Anzahl der Simulatorindizes im Bereich ALLIDX
PARFAC	E+A	1 x unsigned int	I	Simulatorindex der geschnittenen Facility
		1 x signed int	{-1,2..65536}	Bitbreite: -1 = Netz, 2..65536 = Vektor mit angegebener Anzahl von Strands (Vektor der Länge 1 = Netz)
		1 x unsigned char	M	Anzahl der Teilmodelle, die diese Facility speisen (für 'E') bzw. die von dieser Facility gespeist werden (für 'A')
		M x unsigned char	NR	Nummern dieser Teilmodelle (beginnend mit 1)

Tabelle A.4: Neues Format der pmod-Files für die Slaves (1. Teil)

Bereich	Anzahl	Datentyp	Inhalt	Bedeutung
COMFAC	T	1 x unsigned int	Y	Anzahl der eingangsseitig geschnittenen Facilities die von Teilmodell t kommen (t = 1..T)
		Y x unsigned int	U	Index der zugehörigen PARFAC-Struktur
COMFAC	T	1 x unsigned int	Z	Anzahl der ausgangsseitig geschnittenen Facilities die zu Teilmodell t führen (t = 1..T)
		Z x unsigned int	V	Index der zugehörigen PARFAC-Struktur
ALLIDX	F	1 x unsigned int	S	Liste aller Simulator-indizes des originalen Modells. Für Facilities die nicht im Teilmodell vorkommen ist der Simulatorindex = 0.

Tabelle A.5: Neues Format der pmod-Files für die Slaves (2. Teil)

A.2 Klartextversionen der Listenfiles

A.2.1 Klartextversion der Crossreferenzliste (.pxref)

Eintrag	Bedeutung
@T 00 35 35	Erstellungszeit des xref-Files
@D 05 06 00	Erstellungsdatum des xref-Files
@Z 3	Anzahl der Teilmodelle
@X 25764	Anzahl der nachfolgenden Netze und Vektoren
@N BAM_BUSY 1 3	Netzname, Anzahl der Teilmodelle in denen sich das Netz befindet (1), Nummern dieser Teilmodelle (3)
@N BAM_YARTRY 3 1 2 3	Dieses Netz befindet sich in 3 Teilmodellen, die Teilmodell-nummern sind 1, 2 und 3.
:	:
@V BIO_BRST_CNTL(0..3) 3 1 2 3	Dies ist ein ungesplitteter Vektor: Die Strands 0 bis 3 befinden sich in allen 3 Teilmodellen (1,2,3).
:	:
@N ZMB_TT(0) 3 1 2 3	Dies ist ein aufgesplitteter Vektor:
@N ZMB_TT(1) 3 1 2 3	Die Strands 0 bis 3 befinden sich
@N ZMB_TT(2) 3 1 2 3	in den Teilmodellen 1, 2 und 3.
@N ZMB_TT(3) 3 1 2 3	Aber Strand 4 befindet sich nur
@N ZMB_TT(4) 2 1 3	in den Teilmodellen 1 und 3. In diesem Fall erscheinen alle Vektorstrands des Vektors in der Liste als separate Netze.
:	:
@Y 116	Anzahl der nachfolgenden Arrays
@A BAD_.BG0.XGDA.AD 1 1	Dieses Array befindet sich in Teilmodell 1.
@A CPU0.LXA_.XC0.XCAA.AD 2 1 2	Dieses Array befindet sich in den Teilmodellen 1 und 2.
:	:

Tabelle A.6: Format des xref-Files für den Master (unverändert)

A.2.2 Klartextversion der Signalschnittliste (.ext)

Eintrag	Bedeutung
@T 00 35 35	Erstellungszeit des ext-Files
@D 05 06 00	Erstellungsdatum des ext-Files
@X 1240	Anzahl der nachfolgenden eingangsseitig geschnittenen Facilities
@I SOP_.BOP.NOA.L 1 3	Diese eingangsseitig geschnittene Facility ist ein Netz und kommt von Teilmodell 3.
@I CPU0.LXA_.XDD.XDDI.L(4) 1 2	Diese eingangsseitig geschnittene Facility ist ein Vektor mit 4 Strands. Der Vektor kommt von Teilmodell 2.
:	:
@Y 524	Anzahl der nachfolgenden ausgangsseitig geschnittenen Facilities
@O BAD_.BAM.XFR_VAL.L 2 2 3	Diese ausgangsseitig geschnittene Facility ist ein Netz und führt in die Teilmodelle 2 und 3.
@O BAD_.BAM.XMAD0.L(16) 2 2 3	Diese ausgangsseitig geschnittene Facility ist ein Vektor mit 16 Strands und führt in die Teilmodelle 2 und 3.
:	:

Tabelle A.7: Format der neuen ext-Files für die Slaves

Anhang B

Die Funktionen des neuen PS-API's

void ps_cycle(void)
Diese Funktion erzwingt die Evaluierung der Logik eines Teilmodells. Sie wird als Ersatz für das direkte Auslesen eines Signals im Rahmen der automatischen Evaluierung aller Teilmodelle (Stichwort: *Autoevaluation*, siehe [LÖSER 1998], S. 51) eingesetzt.

int ps_facdatalen(int facidx, int offset, int length)
Diese Funktion ermittelt die Anzahl der Bytes, die zur Speicherung des Wertes der übergebenen Facility benötigt werden. Sie wird eingesetzt, um den von einer geschnittenen Facility im Kommunikationsvektor benötigten Speicherplatz zu bestimmen.

int ps_getfacdata(char *buff, int facidx, int offset, int length)
Diese Funktion ermittelt den Wert der übergebenen Facility-Referenz und kopiert diesen nach *buff*. Der Rückgabewert enthält die Anzahl der geschriebenen Bytes. *ps_getfacdata()* wird im GET-Schritt des parallelen Clock-Cycle-Algorithmus zum Auslesen der Werte der geschnittenen Facilities verwendet.

int ps_putfacdata(char *buff, int facidx, int offset, int length)
Diese Funktion setzt den Wert der übergebenen Facility auf den Datenwert in *buff*. Der Rückgabewert enthält die Anzahl der aus *buff* ausgelesenen Bytes. Verwendet wird die Funktion im PUT-Schritt eines parallelen Clock-Cycles zum Zurückschreiben der Werte der geschnittenen Facilities in die Teilmodelle.

Anhang C

Verteilung der aktiven Teilmodelle in den Experimenten zur dynamischen Lastbalancierung

C.1 Experiment „No Load“

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
1	1000	1	3	2	1	2	3	2	3
1001	2000	1	3	2	1	2	3	2	3
2001	3000	1	1	2	1	1	3	2	3
3001	4000	1	1	2	1	1	3	2	3
4001	5000	1	1	1	1	1	3	2	2
5001	6000	1	1	1	1	1	3	2	2
6001	7000	1	1	1	1	1	1	2	3
7001	8000	1	1	1	1	1	1	2	3
8001	30000	1	1	1	3	1	1	2	3

Tabelle C.1: Experiment „No Load“

C.2 Experiment „Load 2“

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
1	1000	1	3	2	1	2	3	2	3
1001	2000	1	3	2	1	2	3	2	3
2001	3000	1	2	1	1	1	3	2	3
3001	4000	1	2	1	1	1	3	2	3
4001	5000	1	1	1	1	1	3	2	2
5001	6000	1	1	1	1	1	3	2	2
6001	7000	1	1	1	1	1	1	2	3
7001	8000	1	1	1	1	1	1	2	3
8001	9000	1	1	1	3	1	1	2	3
9001	10000	1	1	1	3	1	1	2	3
10001	11000	1	1	1	3	1	1	2	3
11001	12000	1	1	1	3	1	1	2	3
12001	13000	2	3	1	3	1	1	2	3
13001	14000	2	3	1	3	1	1	2	3
14001	15000	2	3	1	3	1	1	2	3
15001	16000	2	3	1	3	1	1	2	3
16001	17000	2	3	1	3	1	1	2	3
17001	18000	2	3	1	3	1	1	2	3
18001	19000	2	3	1	3	1	1	2	3
19001	20000	2	3	1	3	1	1	2	3
20001	21000	2	3	1	3	1	1	2	3
21001	22000	2	3	1	3	1	1	2	3
22001	23000	1	3	2	1	1	1	2	3
23001	24000	1	3	2	1	1	1	2	3
24001	25000	1	3	1	1	1	1	2	3
25001	26000	1	3	1	1	1	1	2	3
26001	27000	1	1	1	1	1	3	2	3
27001	28000	1	1	1	1	1	3	2	3
28001	29000	1	1	1	3	1	1	2	3
29001	30000	1	1	1	3	1	1	2	3

Tabelle C.2: Experiment „Load 2“

C.3 Experiment „Load 4“

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
1	1000	1	3	2	1	2	3	2	3
1001	2000	1	3	2	1	2	3	2	3
2001	3000	1	2	1	1	1	3	2	3
3001	4000	1	2	1	1	1	3	2	3
4001	5000	1	1	1	1	1	3	2	2
5001	6000	1	1	1	1	1	3	2	2
6001	7000	1	1	1	1	1	1	2	3
7001	8000	1	1	1	1	1	1	2	3
8001	9000	1	1	1	3	1	1	2	3
9001	10000	1	1	1	3	1	1	2	3
10001	11000	1	1	1	3	1	1	2	3
11001	12000	1	1	1	3	1	1	2	3
12001	13000	2	2	1	3	1	3	2	3
13001	14000	2	2	1	3	1	3	2	3
14001	15000	2	1	2	3	2	3	2	3
15001	16000	2	1	2	3	2	3	2	3
16001	17000	1	2	2	3	2	3	2	3
17001	18000	1	2	2	3	2	3	2	3
18001	19000	1	3	2	3	2	1	2	3
19001	20000	1	3	2	3	2	1	2	3
20001	21000	1	3	2	3	2	1	2	3
21001	22000	1	3	2	3	2	1	2	3
22001	23000	1	3	2	1	1	1	2	3
23001	24000	1	3	2	1	1	1	2	3
24001	25000	1	3	1	1	1	1	2	3
25001	26000	1	3	1	1	1	1	2	3
26001	27000	1	1	1	1	1	3	2	3
27001	28000	1	1	1	1	1	3	2	3
28001	29000	1	1	1	3	1	1	2	3
29001	30000	1	1	1	3	1	1	2	3

Tabelle C.3: Experiment „Load 4“

C.4 Experiment „Load 1 on W_3“

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
1	1000	1	3	2	1	2	3	2	3
1001	2000	1	3	2	1	2	3	2	3
2001	3000	1	2	1	1	1	3	2	3
3001	4000	1	2	1	1	1	3	2	3
4001	5000	1	1	1	1	1	3	2	2
5001	6000	1	1	1	1	1	3	2	2
6001	7000	1	1	1	1	1	1	2	3
7001	8000	1	1	1	1	1	1	2	3
8001	9000	1	1	1	3	1	1	2	3
9001	10000	1	1	1	3	1	1	2	3
10001	11000	1	1	1	3	1	1	2	3
11001	12000	1	1	1	3	1	1	2	3
12001	13000	1	1	1	1	1	1	2	3
13001	14000	1	1	1	1	1	1	2	3
14001	15000	1	1	1	1	1	1	2	3
15001	16000	1	1	1	1	1	1	2	3
16001	17000	1	1	1	1	1	1	2	3
17001	18000	1	1	1	1	1	1	2	3
18001	19000	1	1	1	1	1	1	2	3
19001	20000	1	1	1	1	1	1	2	3

Tabelle C.4: Experiment „Load 1 on W_3“, Teil 1

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
20001	21000	1	1	1	1	1	1	2	3
21001	22000	1	1	1	1	1	1	2	3
22001	23000	1	1	1	3	1	1	2	3
23001	24000	1	1	1	3	1	1	2	3
24001	25000	1	1	1	3	1	1	2	3
25001	26000	1	1	1	3	1	1	2	3
26001	27000	1	1	1	3	1	1	2	3
27001	28000	1	1	1	3	1	1	2	3
28001	29000	1	1	1	3	1	1	2	3
29001	30000	1	1	1	3	1	1	2	3
30001	31000	1	1	1	3	1	1	2	3
31001	32000	1	1	1	3	1	1	2	3
32001	33000	1	1	1	3	1	1	2	3
33001	34000	1	1	1	3	1	1	2	3
34001	35000	1	1	1	3	1	1	2	3
35001	36000	1	1	1	3	1	1	2	3
36001	37000	1	1	1	3	1	1	2	3
37001	38000	1	1	1	3	1	1	2	3
38001	39000	1	1	1	3	1	1	2	3
39001	40000	1	1	1	3	1	1	2	3

Tabelle C.5: Experiment „Load 1 on W_3“, Teil 2

C.5 Experiment „Load 2 on W_3“

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
1	1000	1	3	2	1	2	3	2	3
1001	2000	1	3	2	1	2	3	2	3
2001	3000	1	2	1	1	1	3	2	3
3001	4000	1	2	1	1	1	3	2	3
4001	5000	1	1	1	1	1	3	2	2
5001	6000	1	1	1	1	1	3	2	2
6001	7000	1	1	1	1	1	1	2	3
7001	8000	1	1	1	1	1	1	2	3
8001	9000	1	1	1	3	1	1	2	3
9001	10000	1	1	1	3	1	1	2	3
10001	11000	1	1	1	3	1	1	2	3
11001	12000	1	1	1	3	1	1	2	3
12001	13000	1	1	1	1	1	1	2	2
13001	14000	1	1	1	1	1	1	2	2
14001	15000	1	1	1	1	1	1	2	2
15001	16000	1	1	1	1	1	1	2	2
16001	17000	1	1	1	1	1	1	2	2
17001	18000	1	1	1	1	1	1	2	2
18001	19000	1	1	1	1	1	1	2	2
19001	20000	1	1	1	1	1	1	2	2

Tabelle C.6: Experiment „Load 2 on W_3“, Teil 1

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
20001	21000	1	1	1	1	1	1	2	2
21001	22000	1	1	1	1	1	1	2	2
22001	23000	1	1	1	1	1	1	2	2
23001	24000	1	1	1	3	1	3	3	2
24001	25000	1	1	1	3	1	3	3	2
25001	26000	1	1	2	1	1	1	3	2
26001	27000	1	1	2	1	1	1	3	2
27001	28000	2	1	1	1	1	1	3	2
28001	29000	2	1	1	1	1	1	3	2
29001	30000	1	2	1	1	1	1	3	2
30001	31000	1	2	1	1	1	1	3	2
31001	32000	2	1	1	1	1	1	3	2
32001	33000	2	1	1	1	1	1	3	2
33001	34000	1	2	1	3	1	1	3	2
34001	35000	1	2	1	3	1	1	3	2
35001	36000	1	1	2	3	1	1	3	2
36001	37000	1	1	2	3	1	1	3	2
37001	38000	1	1	1	3	1	1	3	2
38001	39000	1	1	1	3	1	1	3	2
39001	40000	1	1	1	3	1	1	3	2

Tabelle C.7: Experiment „Load 2 on W_3“, Teil 2

C.6 Experiment „Load 2 on W_2“

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
1	1000	1	3	2	1	2	3	2	3
1001	2000	1	3	2	1	2	3	2	3
2001	3000	1	1	2	1	1	3	2	3
3001	4000	1	1	2	1	1	3	2	3
4001	5000	1	1	1	1	1	3	2	2
5001	6000	1	1	1	1	1	3	2	2
6001	7000	1	1	1	1	1	1	2	3
7001	8000	1	1	1	1	1	1	2	3
8001	9000	1	1	1	3	1	1	2	3
9001	10000	1	1	1	3	1	1	2	3
10001	11000	1	1	1	3	1	1	2	3
11001	12000	1	1	1	3	1	1	2	3
12001	13000	1	1	1	3	1	1	2	3
13001	14000	1	1	1	3	1	1	2	3
14001	15000	1	1	1	3	1	1	2	3
15001	16000	1	1	1	3	1	1	2	3
16001	17000	1	1	1	3	1	1	2	3
17001	18000	1	1	1	3	1	1	2	3
18001	19000	1	1	1	3	1	1	2	3
19001	20000	1	1	1	3	1	1	2	3

Tabelle C.8: Experiment „Load 2 on W_2“, Teil 1

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
20001	21000	1	1	1	3	1	1	2	3
21001	22000	1	1	1	3	1	1	2	3
22001	23000	1	1	1	3	1	1	2	3
23001	24000	1	1	1	3	1	1	2	3
24001	25000	1	1	1	3	1	1	2	3
25001	26000	1	1	1	3	1	1	2	3
26001	27000	1	1	1	3	1	1	2	3
27001	28000	1	1	1	3	1	1	2	3
28001	29000	1	1	1	3	1	1	2	3
29001	30000	1	1	1	3	1	1	2	3
30001	31000	1	1	1	3	1	1	2	3
31001	32000	1	1	1	3	1	1	2	3
32001	33000	1	1	1	3	1	1	2	3
33001	34000	1	1	1	3	1	1	2	3
34001	35000	1	1	1	3	1	1	2	3
35001	36000	1	1	1	3	1	1	2	3
36001	37000	1	1	1	3	1	1	2	3
37001	38000	1	1	1	3	1	1	2	3
38001	39000	1	1	1	3	1	1	2	3
39001	40000	1	1	1	3	1	1	2	3

Tabelle C.9: Experiment „Load 2 on W_2“, Teil 2

C.7 Experiment „Simulation interval 500“

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
1	500	1	3	2	1	2	3	2	3
501	1000	1	3	2	1	2	3	2	3
1001	1500	1	1	2	1	1	3	2	3
1501	2000	1	1	2	1	1	3	2	3
2001	2500	1	1	1	3	1	1	2	3
2501	3000	1	1	1	3	1	1	2	3
3001	3500	1	1	1	3	1	1	2	3
3501	4000	1	1	1	3	1	1	2	3
4001	4500	1	1	1	3	1	1	2	3
4501	5000	1	1	1	3	1	1	2	3
5001	5500	1	1	1	3	1	1	2	3
5501	6000	1	1	1	3	1	1	2	3
6001	6500	1	1	1	3	1	1	2	3
6501	7000	1	1	1	3	1	1	2	3
7001	7500	1	1	1	3	1	1	2	3
7501	8000	1	1	1	3	1	1	2	3
8001	8500	1	1	1	3	1	1	2	3
8501	9000	1	1	1	3	1	1	2	3
9001	9500	1	1	1	3	1	1	2	3
9501	10000	1	1	1	3	1	1	2	3
10001	10500	1	1	1	3	1	1	2	3
10501	11000	1	1	1	3	1	1	2	3
11001	11500	2	3	1	3	1	1	2	3
11501	12000	2	3	1	3	1	1	2	3
12001	12500	2	3	1	3	1	1	2	3
12501	13000	2	3	1	3	1	1	2	3
13001	13500	2	3	1	3	1	1	2	3
13501	14000	2	3	1	3	1	1	2	3
14001	14500	2	3	1	3	1	1	2	3
14501	15000	2	3	1	3	1	1	2	3

Tabelle C.10: Experiment „Simulation interval 500“, Teil 1

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
15001	15500	2	3	1	3	1	1	2	3
15501	16000	2	3	1	3	1	1	2	3
16001	16500	2	3	1	3	1	1	2	3
16501	17000	2	3	1	3	1	1	2	3
17001	17500	2	3	1	3	1	1	2	3
17501	18000	2	3	1	3	1	1	2	3
18001	18500	2	3	1	3	1	1	2	3
18501	19000	2	3	1	3	1	1	2	3
19001	19500	2	3	1	3	1	1	2	3
19501	20000	2	3	1	3	1	1	2	3
20001	20500	2	3	1	3	1	1	2	3
20501	21000	2	3	1	3	1	1	2	3
21001	21500	1	3	2	1	1	1	2	3
21501	22000	1	3	2	1	1	1	2	3
22001	22500	1	3	1	1	1	1	2	3
22501	23000	1	3	1	1	1	1	2	3
23001	23500	1	1	1	1	1	3	2	3
23501	24000	1	1	1	1	1	3	2	3
24001	24500	1	1	1	3	1	1	2	3
24501	25000	1	1	1	3	1	1	2	3
25001	25500	1	1	1	3	1	1	2	3
25501	26000	1	1	1	3	1	1	2	3
26001	26500	1	1	1	3	1	1	2	3
26501	27000	1	1	1	3	1	1	2	3
27001	27500	1	1	1	3	1	1	2	3
27501	28000	1	1	1	3	1	1	2	3
28001	28500	1	1	1	3	1	1	2	3
28501	29000	1	1	1	3	1	1	2	3
29001	29500	1	1	1	3	1	1	2	3
29501	30000	1	1	1	3	1	1	2	3

Tabelle C.11: Experiment „Simulation interval 500“, Teil 2

C.8 Experiment „Simulation interval 250“ (Load 4)

Cycle interval		Nodes where given model is active							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
1	250	1	3	2	1	2	3	2	3
251	500	1	3	2	1	2	3	2	3
501	750	1	1	2	1	1	3	2	3
751	1000	1	1	2	1	1	3	2	3
1001	1250	1	1	1	3	1	1	2	3
1251	1500	1	1	1	3	1	1	2	3
1501	1750	1	1	1	3	1	1	2	3
1751	2000	1	1	1	3	1	1	2	3
2001	2250	1	1	1	3	1	1	2	3
2251	2500	1	1	1	3	1	1	2	3
2501	2750	1	1	1	3	1	1	2	3
2751	3000	1	1	1	3	1	1	2	3
3001	3250	1	1	1	3	1	1	2	3
3251	3500	1	1	1	3	1	1	2	3
3501	3750	1	1	1	3	1	1	2	3
3751	4000	1	1	1	3	1	1	2	3
4001	4250	1	1	1	3	1	1	2	3
4251	4500	1	1	1	3	1	1	2	3
4501	4750	1	1	1	3	1	1	2	3
4751	5000	1	1	1	3	1	1	2	3
5001	5250	1	1	1	3	1	1	2	3
5251	5500	1	1	1	3	1	1	2	3
5501	5750	1	1	1	3	1	1	2	3
5751	6000	1	1	1	3	1	1	2	3
6001	6250	1	1	1	3	1	1	2	3
6251	6500	1	1	1	3	1	1	2	3
6501	6750	1	1	1	3	1	1	2	3
6751	7000	1	1	1	3	1	1	2	3
7001	7250	1	1	1	3	1	1	2	3
7251	7500	1	1	1	3	1	1	2	3

Tabelle C.12: Experiment „Simulation interval 250“ (Load 4), Teil 1

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
7501	7750	1	1	1	3	1	1	2	3
7751	8000	1	1	1	3	1	1	2	3
8001	8250	1	1	1	3	1	1	2	3
8251	8500	1	1	1	3	1	1	2	3
8501	8750	1	1	1	3	1	1	2	3
8751	9000	1	1	1	3	1	1	2	3
9001	9250	1	1	1	3	1	1	2	3
9251	9500	1	1	1	3	1	1	2	3
9501	9750	1	1	1	3	1	1	2	3
9751	10000	1	1	1	3	1	1	2	3
10001	10250	1	1	1	3	1	1	2	3
10251	10500	1	1	1	3	1	1	2	3
10501	10750	2	3	2	3	1	1	2	3
10751	11000	2	3	2	3	1	1	2	3
11001	11250	2	3	2	3	1	1	2	3
11251	11500	2	3	2	3	1	1	2	3
11501	11750	2	3	2	3	1	1	2	3
11751	12000	2	2	2	3	1	1	3	2
12001	12250	2	2	2	3	1	1	3	2
12251	12500	2	2	1	3	2	3	3	2
12501	12750	2	2	1	3	2	3	3	2
12751	13000	2	3	1	1	2	1	3	2
13001	13250	2	3	1	1	2	1	3	2
13251	13500	2	3	1	1	2	1	3	2
13501	13750	3	3	1	1	2	1	2	2
13751	14000	3	3	1	1	2	1	2	2
14001	14250	2	3	1	1	2	1	2	3
14251	14500	2	3	1	1	2	1	2	3
14501	14750	2	3	1	1	2	1	3	2
14751	15000	2	3	1	1	2	1	3	2

Tabelle C.13: Experiment „Simulation interval 250“ (Load 4), Teil 2

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
15001	15250	3	3	1	1	2	1	2	2
15251	15500	3	3	1	1	2	1	2	2
15501	15750	2	3	1	1	2	1	2	3
15751	16000	2	3	1	1	2	1	2	3
16001	16250	2	3	1	1	2	1	3	2
16251	16500	2	3	1	1	2	1	3	2
16501	16750	3	3	1	1	2	1	2	2
16751	17000	3	3	1	1	2	1	2	2
17001	17250	3	2	1	1	2	1	3	2
17251	17500	3	2	1	1	2	1	3	2
17501	17750	2	3	1	1	2	1	3	2
17751	18000	2	3	1	1	2	1	3	2
18001	18250	2	3	1	1	2	1	3	2
18251	18500	3	2	1	1	2	1	3	2
18501	18750	3	2	1	1	2	1	3	2
18751	19000	2	3	1	1	2	1	3	2
19001	19250	2	3	1	1	2	1	3	2
19251	19500	2	3	1	1	2	1	3	2
19501	19750	2	1	1	1	2	3	3	3
19751	20000	2	1	1	1	2	3	3	3
20001	20250	2	1	2	3	2	1	3	3
20251	20500	2	1	2	3	2	1	3	3
20501	20750	2	1	2	1	1	1	3	3
20751	21000	2	1	2	1	1	1	3	3
21001	21250	2	1	1	1	2	1	3	2
21251	21500	2	1	1	1	2	1	3	2
21501	21750	2	1	1	1	1	1	3	2
21751	22000	2	1	1	1	1	1	3	2
22001	22250	1	1	1	3	2	1	3	2
22251	22500	1	1	1	3	2	1	3	2

Tabelle C.14: Experiment „Simulation interval 250“ (Load 4), Teil 3

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
22501	22750	2	1	1	3	1	1	3	2
22751	23000	2	1	1	3	1	1	3	2
23001	23250	1	1	2	3	1	1	3	2
23251	23500	1	1	2	3	1	1	3	2
23501	23750	1	1	1	3	1	1	3	2
23751	24000	1	1	1	3	1	1	3	2
24001	24250	1	1	1	3	1	1	3	2
24251	24500	1	1	1	3	1	1	3	2
24501	24750	1	1	1	3	1	1	3	2
24751	25000	1	1	1	3	1	1	3	2
25001	25250	1	1	1	1	1	1	3	2
25251	25500	1	1	1	1	1	1	3	2
25501	25750	1	1	2	1	1	1	3	2
25751	26000	1	1	2	1	1	1	3	2
26001	26250	2	1	1	1	1	1	3	2
26251	26500	2	1	1	1	1	1	3	2
26501	26750	1	2	1	3	1	1	3	2
26751	27000	1	2	1	3	1	1	3	2
27001	27250	1	1	2	3	1	1	3	2
27251	27500	1	1	2	3	1	1	3	2
27501	27750	1	1	2	3	1	1	3	2
27751	28000	1	1	2	3	1	1	3	2
28001	28250	1	1	2	3	1	1	3	2
28251	28500	1	1	1	3	1	1	3	2
28501	28750	1	1	1	3	1	1	3	2
28751	29000	1	1	1	3	1	1	3	2
29001	29250	1	1	1	3	1	1	3	2
29251	29500	1	1	1	3	1	1	3	2
29501	29750	1	1	1	3	1	1	3	2
29751	30000	1	1	1	3	1	1	3	2

Tabelle C.15: Experiment „Simulation interval 250“ (Load 4), Teil 4

C.9 Experiment „Move offset 2%“

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
1	500	1	3	2	1	2	3	2	3
501	1000	1	3	2	1	2	3	2	3
1001	1500	1	2	1	1	1	3	2	3
1501	2000	1	2	1	1	1	3	2	3
2001	2500	1	1	2	1	1	1	2	3
2501	3000	1	1	2	1	1	1	2	3
3001	3500	1	1	1	1	1	3	2	3
3501	4000	1	1	1	1	1	3	2	3
4001	4500	1	1	1	3	1	1	2	3
4501	5000	1	1	1	3	1	1	2	3
5001	5500	1	1	1	3	1	1	2	3
5501	6000	1	1	1	3	1	1	2	3
6001	6500	1	1	1	3	1	1	2	3
6501	7000	1	1	1	3	1	1	2	3
7001	7500	1	1	1	3	1	1	2	3
7501	8000	1	1	1	3	1	1	2	3
8001	8500	1	1	1	3	1	1	2	3
8501	9000	1	1	1	3	1	1	2	3
9001	9500	1	1	1	3	1	1	2	3
9501	10000	1	1	1	3	1	1	2	3
10001	10500	1	1	1	3	1	1	2	3
10501	11000	1	1	1	3	1	1	2	3
11001	11500	2	3	1	3	1	1	2	3
11501	12000	2	3	1	3	1	1	2	3
12001	12500	2	3	1	3	1	1	2	3
12501	13000	2	3	1	3	1	1	2	3
13001	13500	2	3	1	3	1	1	2	3
13501	14000	2	3	1	3	1	1	2	3
14001	14500	2	3	1	3	1	1	2	3
14501	15000	2	3	1	3	1	1	2	3

Tabelle C.16: Experiment „Move offset 2%“, Teil 1

Cycle interval		**Nodes where given model is active**							
Begin	End	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
15001	15500	2	3	1	3	1	1	2	3
15501	16000	2	3	1	3	1	1	2	3
16001	16500	2	3	1	3	1	1	2	3
16501	17000	2	3	1	3	1	1	2	3
17001	17500	2	3	1	3	1	1	2	3
17501	18000	2	3	1	3	1	1	2	3
18001	18500	2	3	1	3	1	1	2	3
18501	19000	2	3	1	3	1	1	2	3
19001	19500	2	3	1	3	1	1	2	3
19501	20000	2	3	1	3	1	1	2	3
20001	20500	2	3	1	3	1	1	2	3
20501	21000	2	3	1	3	1	1	2	3
21001	21500	1	3	2	1	1	1	2	3
21501	22000	1	3	2	1	1	1	2	3
22001	22500	3	1	1	1	1	1	2	3
22501	23000	3	1	1	1	1	1	2	3
23001	23500	1	1	1	1	1	3	2	3
23501	24000	1	1	1	1	1	3	2	3
24001	24500	1	1	1	3	1	1	2	3
24501	25000	1	1	1	3	1	1	2	3
25001	25500	1	1	1	3	1	1	2	3
25501	26000	1	1	1	3	1	1	2	3
26001	26500	1	1	1	3	1	1	2	3
26501	27000	1	1	1	3	1	1	2	3
27001	27500	1	1	1	3	1	1	2	3
27501	28000	1	1	1	3	1	1	2	3
28001	28500	1	1	1	3	1	1	2	3
28501	29000	1	1	1	3	1	1	2	3
29001	29500	1	1	1	3	1	1	2	3
29501	30000	1	1	1	3	1	1	2	3

Tabelle C.17: Experiment „Move offset 2%“, Teil 2

Literaturverzeichnis

[PEd 1996] (1996). *IBM Parallel Environment for AIX: Operation and Use, Volume 1: „Using the Parallel Operating Environment“*. IBM Corporation.

[Bergman et al. 1999] Bergman, Stephen C., R. Shadowen, Z. Hidvegi, C. Gabele und M. Sustik (1999). *MVLSIM Multivalue Simulation System User's Guide*. IBM Server Division, Austin, TX, IBM Internal Use Only.

[Döhler 1996] Döhler, Denis (1996). *Entwurf und Implementierung eines parallelen Logiksimulators auf der Basis von TEXSIM*. Diplomarbeit, Universität Leipzig, Institut für Informatik.

[Hennings 1998] Hennings, Hilmar (1998). *SP1/SP2-Systeme von IBM*. Problemseminar Parallelrechner II, Universität Leipzig, Institut für Informatik.

[Hennings 1999] Hennings, Hilmar (1999). *Entwurf und Implementierung der Komponente parallelMAP zur Modellanalyse und -partitionierung im Kontext von parallelTEXSIM*. Diplomarbeit, Universität Leipzig, Institut für Informatik.

[Hering et al. 1995] Hering, Klaus, R. Haupt und T. Villmann (1995). *Cone-basierte, hierarchische Modellpartitionierung zur parallelen compilergesteuerten Logiksimulation beim VLSI-Design*. Technical Report 13, Universität Leipzig, Institut für Informatik.

[Hering et al. 1999] Hering, Klaus, H. Hennings und R. Haupt (1999). *DRIVE: A distributed environment supporting combination of sequential and parallel modules*. In: *Proc. of the IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS)*.

[HERING et al. 2001] HERING, KLAUS, J. LÖSER und J. MARKWARDT (2001). *dlbSIM - A Parallel Functional Logic Simulator Allowing Dynamic Load Balancing*. In: *Proc. of the International Conference on Design, Automation and Test in Europe (DATE)*, S. 472–478.

[HIDVEGI und SHADOWEN 2000] HIDVEGI, ZOLTAN und R. SHADOWEN (2000). *MVLLIB function interface, Revision 1.4*. IBM Server Division, Austin, TX, IBM Internal Use Only.

[LÖSER 1998] LÖSER, JORK (1998). *Dynamische Lastbalancierung bei der parallelen Logiksimulation*. Diplomarbeit, Universität Leipzig, Institut für Informatik.

[REILEIN 1998] REILEIN, ROBERT (1998). *Modellpartitionierung zur parallelen Logiksimulation*. Diplomarbeit, Universität Leipzig, Institut für Informatik.

[SCHULZE 1998] SCHULZE, HENDRIK (1998). *Entwicklung, Untersuchung und Implementierung von parallelen Evolutionären Algorithmen für die Modellpartitionierungskomponente parallelMAP*. Diplomarbeit, Universität Leipzig, Institut für Informatik.

[SIEDSCHLAG 1998] SIEDSCHLAG, THOMAS (1998). *Iterative Modellpartitionierungsverfahren für die parallele Logiksimulation*. Diplomarbeit, Universität Leipzig, Institut für Informatik.

[ZIKE 1992] ZIKE, DAVID S. (1992). *Design Automation Data Base (DA-DB)*. Advanced Workstations Division, Austin, TX, IBM Internal Use Only.

[ZIKE 1996] ZIKE, DAVID S. (1996). *Cycle-based Simulation in IBM*. Lecture in Workshop „Parallel Logic Simulation“, Universität Leipzig, Institut für Informatik.

Ich versichere, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, 9. Mai 2001